A COMPILER WRITING SYSTEM

Bilal Ahmed Qureshi

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

March 1979

© Bilal Ahmed Qureshi, 1979

ABSTRACT

A COMPILER WRITING SYSTEM

Bilal Ahmed Qureshi

This thesis presents extensions to the XPL compiler writing system [MK1] to facilitate error repair during parsing, paragraphing the source program, and semantic specification in terms of a high-level abstract machine.

A technique for automatic generation of error-repair tables is also given; this technique applies to all parsing methods. This error-repair technique is used in the scan phase, and can be used in conjunction with the error recovery in the parse phase. We have inserted these algorithims into a SLR(1) parser generator, which, along with a modified XPL skelton compiler, can be used to produce compilers which have good error-recovery capabilities.

Also investigated are techniques for paragraphing, to expose the logical structure of the program input to the compiler, and to improve the readability of the output listing.

This study also discusses techniques concerned with how semantic specifications can be associated with the syntactic specifications of a language, in order to generate synthesis procedures.

ACKNOWLEDGEMENTS

The writer gratefully acknowledges the guidance and assistance of Prof. J. W. Atwood of Concordia University, and Prof. A. Rudmik, Department of Computing and Information Science, Queen's University, which enabled the successful completion of this study.

Also deserving of much thanks is Mr. Larry Thiel, Senior Analyst, Department of Computer Science, Concordia University, for his constructive criticism of this work.

Finally, the writer wishes to thank his many colleagues in the Department of Computer Science at Concordia University for their many useful suggestions.

This work has been supported by funds from the National Research Council of Canada.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This thesis describes extensions to the XPL compiler writing system [MK1] which simplify the use of the system as well as increasing its capabilities. The extended system includes an algorithm for a table driven error-repair scheme and a generator for the error-repair tables and the synthesis procedures. Techniques for compiler-controlled paragraphing are also investigated.

Compiler writing systems are useful tools for implementing compilers. Such systems reduce the implementation cost of a compiler and permit the implementation of compilers with less effort. Language designers can use a compiler writing system in experimenting with language design, to determine the impact of design decisions on actual programmers in a fairly short period of time.

The ability of a compiler writing system to generate a compiler in an automatic way has obvious advantages in an educational environment, where several specialized pedagogical languages, as well as commercially supported languages, are taught.

This thesis addresses the following aspects of compiler writing systems:

CHAPTER 1

INTRODUCTION

This thesis describes extensions to the XPL compiler writing system [MK1] which simplify the use of the system as well as increasing its capabilities. The extended system includes an algorithm for a table driven error-repair scheme and a generator for the error-repair tables and the synthesis procedures. Techniques for compiler-controlled paragraphing are also investigated.

Compiler writing systems are useful tools for implementing compilers. Such systems reduce the implementation cost of a compiler and permit the implementation of compilers with less effort. Language designers can use a compiler writing system in experimenting with language design, to determine the impact of design decisions on actual programmers in a fairly short period of time.

The ability of a compiler writing system to generate a compiler in an automatic way has obvious advantages in an educational environment, where several specialized pedagogical languages, as well as commercially supported languages, are taught.

This thesis addresses the following aspects of compiler writing systems:

- Error Repair
- Paragraphing
- Semantics

Error Repair: Error repair is concerned with repairing syntax errors so as to permit continuation of the compilation process. This thesis presents an error-repair filter which fits in between the scanner and the parser in the compiler. This filter checks the syntactical validity of any two consecutive symbols in the input program. If this given pair is syntactically correct, then the symbols are passed on to the parser; if not, the given symbol pair is repaired first, before being passed on to the parser.

The compiler writer or the language designer utilizing this technique can specify one of the four following actions by which the error will be repaired: Firstly, to skip the second symbol in the given pair; secondly, to insert a given terminal symbol between the first and the second symbol; thirdly, to replace the first symbol of the pair by a given terminal symbol; fourthly, to replace the second symbol of the pair by a given terminal symbol. It must be understood that the above-mentioned terminal symbols must be duly specified by the language designer in the course of generating the compiler. This technique allows the language designer complete control over error repair through the above-mentioned four steps. If none of the steps are specified, then the

- Error Repair
- Paragraphing
- Semantics

Error Repair: Error repair is concerned with repairing syntax errors so as to permit continuation of the compilation process. This thesis presents an error-repair filter which fits in between the scanner and the parser in the compiler. This filter checks the syntactical validity of any two consecutive symbols in the input program. If this given pair is syntactically correct, then the symbols are passed on to the parser; if not, the given symbol pair is repaired first, before being passed on to the parser.

The compiler writer or the language designer utilizing this technique can specify one of the four following actions by which the error will be repaired: Firstly, to skip the second symbol in the given pair; secondly, to insert a given terminal symbol between the first and the second symbol; thirdly, to replace the first symbol of the pair by a given terminal symbol; fourthly, to replace the second symbol of the pair by a given terminal symbol. It must be understood that the above-mentioned terminal symbols must be duly specified by the language designer in the course of generating the compiler. This technique allows the language designer complete control over error repair through the above-mentioned four steps. If none of the steps are specified, then the default action is to skip the second symbol of the syntactically incorrect pair of symbols.

This technique offers a major advantage insofar as it can be used in conjunction with different parsing techniques, thereby making it a highly adaptable approach. In addition, the parser error recovery can be used to augment the error handling capabilities of this technique.

Paragraphing: The basic aim of paragraphing, as dealt with in this study, is to display the logical structure of the input program as well as to improve its readability on the printed page. Of great importance, in this respect, is the use of paragraphing in a padagogical environment. When a student becomes accustomed to seeing his program paragraphed, it helps him to closely focus upon the relationships between program structure (emphasized by paragraphing), program construction and program execution. The paragraphing technique proposed in this thesis rebuilds the entire input text from the symbols produced by the scanner and, if necessary, repaired by the proposed error repair technique.

The proposed paragraphing method has adopted some standard rules for displaying the logical structure of the input programs, which involve indenting logically subordinate code and putting each statement on a new line, etc. This technique handles some special cases differently to improve the readability of the input program. These cases include macros, default action is to skip the second symbol of the syntactically incorrect pair of symbols.

This technique offers a major advantage insofar as it can be used in conjunction with different parsing techniques, thereby making it a highly adaptable approach. In addition, the parser error recovery can be used to augment the error handling capabilities of this technique.

Paragraphing: The basic aim of paragraphing, as dealt with in this study, is to display the logical structure of the input program as well as to improve its readability on the printed page. Of great importance, in this respect, is the use of paragraphing in a padagogical environment. When a student becomes accustomed to seeing his program paragraphed, it helps him to closely focus upon the relationships between program structure (emphasized by paragraphing), program construction and program execution. The paragraphing technique proposed in this thesis rebuilds the entire input text from the symbols produced by the scanner and, if necessary, repaired by the proposed error repair technique.

The proposed paragraphing method has adopted some standard rules for displaying the logical structure of the input programs, which involve indenting logically subordinate code and putting each statement on a new line, etc. This technique handles some special cases differently to improve the readability of the input program. These cases include macros, comments, declare statements, etc. In brief, the macros are not expanded, the position of the comments with respect to the statement is preserved, and, if a list of variables is declared in one declare statement, then each variable is listed on a separate line and they are aligned, and so on.

Semantics: The proposed compiler writing system also generates the synthesis procedures. To generate synthesis procedures, one has to associate the semantic specification with the syntactic specification of the language. The semantics are specified in HDG [RU1] notation, while the syntax is specified in BNF [KN1].

1.1 THESIS OUTLINE

This thesis contains 5 chapters and two appendices. Chapter 2 presents a brief historical review of compiler writing systems. Chapter 3 describes the proposed compiler writing system. Chapter 4 presents the use of the XPL skeleton compiler and discussion of results. Chapter 5 contains the summary and conclusions. Appendix A gives details on how to use the SLR(1) analyzer. Appendix B contains some examples of test programs which were structured and syntactically debugged by a compiler generated by the compiler writing system proposed in this thesis.

CHAPTER 2

HISTORICAL REVIEW

2.1 FORMAL DESCRIPTION OF LANGUAGES

A language, be it natural, formal, or artificial, has two aspects: syntax and semantics.

The syntactic specification of a context-free language, being simply the specification of the permissible sequences of words, or tokens, in the language, has been possible using Backus Naur form for several years [BA1]. Other forms, such as graphs, have also been postulated [FE2], but all of these methods of specifying syntax rely on the fact that syntax is an abstract concept; it is a series of rules for forming sequences of tokens, or, conversely, for recognizing sequences of tokens. The tokens themselves are simply labels and the syntax of a language is independent of the entities these labels represent. These entities are the semantics of the language [GO1].

Semantics in general has not been rigorously definable and, indeed, it can not be due to its dependence on some ultimate reference that has not yet been agreed upon. Certainly the adoption of a Universal Machine which could perform any describable task in a well-defined and unambiguous way, and whose description was equally rigorous, would permit

CHAPTER 2

HISTORICAL REVIEW

2.1 FORMAL DESCRIPTION OF LANGUAGES

A language, be it natural, formal, or artificial, has two aspects: syntax and semantics.

The syntactic specification of a context-free language, being simply the specification of the permissible sequences of words, or tokens, in the language, has been possible using Backus Naur form for several years [BA1]. Other forms, such as graphs, have also been postulated [FE2], but all of these methods of specifying syntax rely on the fact that syntax is an abstract concept; it is a series of rules for forming sequences of tokens, or, conversely, for recognizing sequences of tokens. The tokens themselves are simply labels and the syntax of a language is independent of the entities these labels represent. These entities are the semantics of the language [GO1].

Semantics in general has not been rigorously definable and, indeed, it can not be due to its dependence on some ultimate reference that has not yet been agreed upon. Certainly the adoption of a Universal Machine which could perform any describable task in a well-defined and unambiguous way, and whose description was equally rigorous, would permit a linguist to describe semantics in a frame understood by all. A Turing machine would do for the computer scientist. However, such a machine is impractical, to say the least, and semantic specifications for formal languages must resort to appeals to global concepts expressed in a natural language. For example, the Algol 68 report gives semantics for the syntax <integer> ::= <integer> + <integer> as the result obtained "in the sense of numerical analysis".

An alternative approach has been to select a well-known formal language (such as Algol) to define the semantics of statements in a new language. This technique depends on the fact that the semantics of the base language are well-defined, but this is not always the case. (In Algol, for example, the semantic action for the addition of a character string and a number is dependent on the compiler used).

No matter what approach to semantic specifications is used, it must be remembered that semantics are well-defined only when related to the actions of some predictable machine (be ~~it~~ real or pseudo) and, therefore, in specifying semantics, one must first specify the device which will execute the semantics [G01].

2.2 COMPILERS

In the beginning, there was machine language. It was not long before computer linguists were writing assemblers to make a linguist to describe semantics in a frame understood by all. A Turing machine would do for the computer scientist. However, such a machine is impractical, to say the least, and semantic specifications for formal languages must resort to appeals to global concepts expressed in a natural language. For example, the Algol 68 report gives semantics for the syntax <integer> ::= <integer> + <integer> as the result obtained "in the sense of numerical analysis".

An alternative approach has been to select a well-known formal language (such as Algol) to define the semantics of statements in a new language. This technique depends on the fact that the semantics of the base language are well-defined, but this is not always the case. (In Algol, for example, the semantic action for the addition of a character string and a number is dependent on the compiler used).

No matter what approach to semantic specifications is used, it must be remembered that semantics are well-defined only when related to the actions of some predictable machine (be ~~it~~ real or pseudo) and, therefore, in specifying semantics, one must first specify the device which will execute the semantics [G01].

2.2 COMPILERS

In the beginning, there was machine language. It was not long before computer linguists were writing assemblers to make source programs easier to code, and such a philosophy led naturally to the creation of high-level languages and their compilers. While the early compilers all adopted ad hoc techniques for translating from a particular source language to a particular object language, the formalization of language syntax specifications permitted various parsing techniques to be developed and used in compiler writing. Included here are such things as table-driven parsers, which were used in many early compilers and are still being revisited [FE1], the stack algorithms of Irons [IR1], the transition matrices of Gries [GR1], and recently the deterministic pushdown automata (DPDA) parsers of DeRemer [DR1].

However, prior to the advent of compiler writing systems (discussed below), all of these techniques required the manual creation of parsing tables by the compiler writer. Further, symbol table organization and algorithms, although categorized, were invariably coded by each compiler writer, as were the scan routines and code generators and emitters. Each major language spawned a variety of dialects and special compilers. There were standard FORTRAN compilers, fast FORTRAN compilers, diagnostic FORTRAN compilers, and optimizing FORTRAN compilers. Many seemed reconciled to this myriad of languages and compilers, for to devise a new language and then to write its compiler was beyond the scope of anyone not prepared to invest at least a man-year of work (unless he could suffice with modifying an existing compiler).

source programs easier to code, and such a philosophy led naturally to the creation of high-level languages and their compilers. While the early compilers all adopted ad hoc techniques for translating from a particular source language to a particular object language, the formalization of language syntax specifications permitted various parsing techniques to be developed and used in compiler writing. Included here are such things as table-driven parsers, which were used in many early compilers and are still being revisited [FE1], the stack algorithms of Irons [IR1], the transition matrices of Gries [GR1], and recently the deterministic pushdown automata (DPDA) parsers of DeRemer [DR1].

However, prior to the advent of compiler writing systems (discussed below), all of these techniques required the manual creation of parsing tables by the compiler writer. Further, symbol table organization and algorithms, although categorized, were invariably coded by each compiler writer, as were the scan routines and code generators and emitters. Each major language spawned a variety of dialects and special compilers. There were standard FORTRAN compilers, fast FORTRAN compilers, diagnostic FORTRAN compilers, and optimizing FORTRAN compilers. Many seemed reconciled to this myriad of languages and compilers, for to devise a new language and then to write its compiler was beyond the scope of anyone not prepared to invest at least a man-year of work (unless he could suffice with modifying an existing compiler).

2.3 COMPILER WRITING SYSTEMS

Since compiler writing is a large programming task with many aspects, any system feature which helps one produce pieces of a compiler is a compiler writing tool. The work on automated syntax analysis methods is the oldest and best understood aspect of compiler writing system research.

Historically, the existence of compiler writing systems (or, compiler compilers) is a result of using syntax-directed compiling techniques in order to structure the compiler. Syntax becomes a language in which parts of a compiler may be written, and the concept is extended to semantics by including a compatible programming language, usually general purpose, which allows classical programming methods for those parts of the compiler not susceptible to treatment by syntax. Compiler compilers depend very much on language definition techniques since, for example, it is difficult to use syntax-directed methods for a language which has no syntax [GR1].

A compiler compiler offers a computer user the opportunity to devise his own language and, by simply specifying its syntax and semantics, to obtain a working compiler for that language. We may consider them satisfactory when it is no longer necessary for a programmer to intervene to achieve this goal [GR1]. The ease with which syntax and semantics may be specified, the diagnostics produced for any ambiguities or

2.3 COMPILER WRITING SYSTEMS

Since compiler writing is a large programming task with many aspects, any system feature which helps one produce pieces of a compiler is a compiler writing tool. The work on automated syntax analysis methods is the oldest and best understood aspect of compiler writing system research.

Historically, the existence of compiler writing systems (or, compiler compilers) is a result of using syntax-directed compiling techniques in order to structure the compiler. Syntax becomes a language in which parts of a compiler may be written, and the concept is extended to semantics by including a compatible programming language, usually general purpose, which allows classical programming methods for those parts of the compiler not susceptible to treatment by syntax. Compiler compilers depend very much on language definition techniques since, for example, it is difficult to use syntax-directed methods for a language which has no syntax [GR1].

A compiler compiler offers a computer user the opportunity to devise his own language and, by simply specifying its syntax and semantics, to obtain a working compiler for that language. We may consider them satisfactory when it is no longer necessary for a programmer to intervene to achieve this goal [GR1]. The ease with which syntax and semantics may be specified, the diagnostics produced for any ambiguities or inconsistencies detected, and the efficiency and (more importantly) the precision of the emitted compiler and the code it will produce, are the criteria for judging a compiler compiler [GO1].

The best known early compiler compiler system is probably that written in Atlas Autocode by Brooker, Morris, et al [BR1]. In this system, the syntax of a language is given in a BNF-like set of productions for the statements of a phrase-structure language, and the semantics is given by a corresponding set of routines which define the machine language to be emitted for a recognized statement. This compiler compiler contains several concepts common to succeeding systems, the obvious ones being the basic format of making syntactic productions the basic elements of the language and associating semantics with them, and the intrinsic extensibility of the system to permit a user to add routines if those supplied are not sufficient. However, it is extremely statement-oriented, with a global symbol table, but with no provision for program structure or lexical levels and the accompanying concepts of scope for symbols. Code emission is sequential as statements are recognized and there exists no parse stack or other such mechanism for subordinating some statements to others, such as in DO groups, IF THEN ELSE sequences, and other now-common constructions. Nevertheless, it is still of considerable academic interest.

By the early 1960's, there was an accompanying array of compilers produced. Considering the amount of effort that has gone into compiler writing, there was a definite need to develop compiler writing systems which could produce efficient compilers. The efficiency of a compiler depends on its ability to conserve time and space, both while translating and during execution of the object program. The efficiency also depends on the error-detection and error-recovery facilities in the compiler.

By that time, as was discussed earlier, several standard techniques of compilation were known, and the task of writing a compiler became a straight-forward though rather tedious operation. Impatience on the part of the programmers, and the ever-lowering cost of hardware (and consequently computer time) relative to the cost of programming, stimulated the adoption of high-level languages for compiler writing systems. This led to the development of complete compiler compilers, such as, 'a processor generator system' by Gorrie [GO1] and 'a compiler writing system' by Lecarme [LE1]. Most of these compiler writing systems are pedagogical tools and sacrifice error-handling capabilities to achieve code efficiency of the generated compiler.

Another compiler compiler is the XPL system created at Stanford University [MK1]. Using XPL, the syntax of the language specified in BNF is fed to a program called ANALYZER which produces a set of parsing tables. These, when combined with a framework (called SKELETON) written in XPL, provides an XPL source program which compiles to produce a recognizer for the specified language. It is the user's responsibility to code in XPL and to add to this recognizer his own scanner, symbol table and associated routines, code synthesizers and code emmiters. While the XPL system offers an automatically generated recognizer with the framework for the rest of the compiler, and a compiler for XPL source that produces efficient IBM/360 machine code, it does stop short of being a proper compiler compiler. It could be argued that this provides desirable freedom in designing the rest of the compiler, especially in the choice of object code emitted, and this is of course true. However, the ability to specify semantics in a high-level meta-language, or as a series of macros, is a necessity if the goal of a compiler system is to be realized: namely, the automatic generation of a complete compiler, given only the specification of the source language.

Such an extension of the XPL project has one possible disadvantage: for a given compiler compiler the target or object language must be fixed to give the basis on which to specify semantics. This restraint is of no consequence, however, if the target language is object code for a high-level pseudo-machine and, in fact, the inclusion of the pseudo-machine emulator in the generated compiler source program yields a complete language processor which, once compiled, will compile and execute source programs written in with a framework (called SKELETON) written in XPL, provides an XPL source program which compiles to produce a recognizer for the specified language. It is the user's responsibility to code in XPL and to add to this recognizer his own scanner, symbol table and associated routines, code synthesizers and code emmiters. While the XPL system offers an automatically generated recognizer with the framework for the rest of the compiler, and a compiler for XPL source that produces efficient IBM/360 machine code, it does stop short of being a proper compiler compiler. It could be argued that this provides desirable freedom in designing the rest of the compiler, especially in the choice of object code emitted, and this is of course true. However, the ability to specify semantics in a high-level meta-language, or as a series of macros, is a necessity if the goal of a compiler system is to be realized: namely, the automatic generation of a complete compiler, given only the specification of the source language.

Such an extension of the XPL project has one possible disadvantage: for a given compiler compiler the target or object language must be fixed to give the basis on which to specify semantics. This restraint is of no consequence, however, if the target language is object code for a high-level pseudo-machine and, in fact, the inclusion of the pseudo-machine emulator in the generated compiler source program yields a complete language processor which, once compiled, will compile and execute source programs written in the specified language. Thus, the proposed compiler writing system in this thesis is the consequence of extending the theories of compiler compilers to conceive of a compiler writing system which can produce compilers with improved error repair and paragraphing capabilities.

2.4 RELATED WORK IN SEMANTICS

A great deal of progress has been made in the last few years towards the development of a theoretical framework appropriate to formal analysis and specification of the semantic aspects of computer languages.

A general language-independent framework of semantic concepts would help to standardize terminology, clarify similarities and differences between languages, and allow rigorous formulation and proof of semantic properties of languages. A language designer could analyse proposed constructs to help find undesirable restrictions, incompatibilities, ambiguities, and so on. A theory of semantics, therefore, should contribute to systematic composition and verification of programs, especially compilers.

Early work in formalization of semantics was done by Feldman [FE1]. He developed a semantic meta-language for representing the meanings of statements in a large class of computer languages. This meta-language is known as Formal the specified language. Thus, the proposed compiler writing system in this thesis is the consequence of extending the theories of compiler compilers to conceive of a compiler writing system which can produce compilers with improved error repair and paragraphing capabilities.

2.4 RELATED WORK IN SEMANTICS

A great deal of progress has been made in the last few years towards the development of a theoretical framework appropriate to formal analysis and specification of the semantic aspects of computer languages.

A general language-independent framework of semantic concepts would help to standardize terminology, clarify similarities and differences between languages, and allow rigorous formulation and proof of semantic properties of languages. A language designer could analyse proposed constructs to help find undesirable restrictions, incompatibilities, ambiguities, and so on. A theory of semantics, therefore, should contribute to systematic composition and verification of programs, especially compilers.

Early work in formalization of semantics was done by Feldman [FE1]. He developed a semantic meta-language for representing the meanings of statements in a large class of computer languages. This meta-language is known as Formal

Semantic Language (FSL). Among the languages for which translators were completed using FSL are Fortran, Algol, Lisp, etc.

Recent work in this area has been done by Tennent [TE1]. He started with the theory of programming language semantics developed by Scott and Strachey [SC1]. This theory was applied to formal language specification. The languages considered in his work were LOOP [ME1] and GEDANKEN [RE1]. The semantic ideas introduced here provide a conceptual framework for formal semantic specification of almost all features of high-level programming languages.

In an effort to specify the semantics of a programming language, a diagrammatic approach has also been used [CO1]. Cordy has presented a concise notation for describing the semantics of a programming language. The notation utilizes semantic charts which, when encoded as tables, lead to a semantic analyzer/synthesizer.

A graphic notation has also been used to specify semantics [RU1]. Rudmik proposed a model of an optimizing compiler that generates a hierarchical directed graph (HDG) representation of the program on which optimizing transformations are performed. The HDG program representation and the model of the optimizing compiler proved to be suitable for compiler writing systems.

The compiler writing system proposed in this thesis uses the HDG notation to specify the semantics of the given language. This is dicussed in section 3.2. A review of HDG's is given in section 3.2.1.

2.5 RELATED WORK IN ERROR REPAIR

Many techniques have been developed for the purpose of handling syntax errors detected in the course of parsing. This section will highlight noted works on this subject; these works focus upon the general applicability of these techniques to the syntax of any given language.

An initial work by Irons [IR1] describes a technique which generates all parsing possibilities of an input string. An error is detected once the parsing process can not proceed past a certain input symbol. At this point a list of potentially acceptable input symbols is generated. The method requires scanning the input string (or text) and discarding all symbols until an acceptable one from the generated list is located. Leinius [LE2] describes a technique whereby the grammatical phrase containing the error is isolated and all possible parsing possibilities are applied to the given phrase until a valid or acceptable parse is found. If none exists within the given phrase, a larger phrase is attempted. James [JA1] drew upon Leinius' approach, but limited the size of the potential phrase to be considered. He also used spelling correction technique described by Morgan [MO1]. Graham and Rhodes [GR1] proposed a variation whereby, before a symbol is stacked, it is checked to ensure that it would not upset the right hand side of the grammatical configuration described by a syntax rule. In the event of an error, the system attempts to apply grammatical rules in the immediate neighbourhood of the error to acquire relevant information prior to attempting repair.

In most cases, syntactic errors are due to omission or wrong use of symbols merely conveying information about the structural properties of the program, such as commas, semicolons, colons, and various kind of brackets. Omission of elements explicitly denoting program activity, such as operands and operators, is rare. Wirth [WI1] uses heuristics to handle these frequent errors intelligently, and is not concerned with the large majority of cases which seldom occur.

SP/k [BA1, HO1] is a compatible subset of the PL/I language that is designed for teaching programming. SP/k is actually a sequence of language subsets called SP/1, SP/2, ..., SP/8. Each subset introduces new programming language constructs while retaining all the contructs of the preceding subsets. One goal for the SP/k procesor was to maximize the useful information given to students about errors in their program. The compiler attempts to isolate and repair a particular error, and then continues processing until another error is found. This approach can diagnose a number of errors in a program on a single run. The nature of each repair is reported to the student in terms of the source program.

Litecky and Davis [LI1] have provided data on Cobol error frequency for correction of errors in student-oriented compilers, improvement of teaching, and changes in the programming language. They identified the types of errors in a pilot study; then, using the 132 error types found, 1,777 errors were classified in 1,400 runs of 73 Cobol students. Error density was high: 20 percent of the types contained 80 percent of the total frequency of errors, which implies high potential effectiveness for software-based correction of Cobol.

In this thesis, we propose a very simple error-repair technique which is based on symbol-pair matching. This method acts as a filter between the scanner and the parser, and is discussed in section 3.3.

2.6 RELATED WORK IN PARAGRAPHING

Most compilers produce source program listings with one line corresponding to one card (or record) of the input program. The listing is a copy of the input program with line numbers on left hand side. Some compilers have the facility to format the input program before it is printed out. This process is known as paragraphing. Some examples of such compilers are IBM's PL/I check-out compiler [IBM], the Project

SUE system language compiler [CL1] and the SP/k compiler [BA1, HO1].

Barnard [BA1] developed a simple technique for paragraphing and implemented it in the SP/k processor. He used syntax charts for describing the programming language. These syntax charts use three basic paragraphing commands. The first command is Indent which causes the left margin to be set one increment to the right. The second is Outdent which causes the left margin to be set one increment to the left. The final command is New Logical Line which causes the current line to be printed out and the new line to be started with the present left margin.

As a result, the SP/k processor always paragraphs students programs. Each statement is placed on a separate line. Statements nested inside compound statements are indented. Nested procedure definitions are indented. Statements that do not fit on a single line have their continuation indented.

Barnard [BA1] points out some weaknesses in his techniques as follows:

1) If the given text has the following form:

```
IF expression1 THEN
   IF expression2 THEN
      statement1
statement2
```

then the internal IF ... THEN will generate a new line, and the containing IF ... THEN will generate an outdent followed by a new line. This will cause printing of an extra blank line:

```
1  IF expression1 THEN
2     IF expression2 THEN
3        statement1
4
5  statement2
```

2) Another problem concerns the placement of the END statement. Suppose it is desired to have the format

```
DO clause;
   {statement list}
END;
```

where the END is at the same level as the corresponding DO. Barnard's technique cannot specify this format. The closest that it is possible to come to is the following:

```
DO clause;
   {statement list}
   END;
```

The paragraphing technique proposed in this thesis is very simple and handles all the features mentioned above. In addition, for purposes of improved readability of the input program, it handles differently certain special cases such as macros, comments, etc. It is discussed in detail in section 3.4.

CHAPTER 3

A COMPILER WRITING SYSTEM

Compiler writing systems are very useful tools for implementing compilers. Such systems reduce the implementation cost of a compiler; therefore, one can implement more compilers with less effort. They are also excellent tools for a language designer. They encourage experimentation in language design, to determine the effects of various design decisions, especially as they permit one to see the results in a fairly short time. There exist several good compiler writing systems, as discussed in chapter 2. This thesis addresses some specific aspects of compiler writing systems, as follows:

- Error Repair
- Paragraphing
- Semantics

For a compiler writing system we need an implementation language which is readable, has a powerful structuring mechanism and has a compiler which generates efficient code. For this purpose we chose the XPL language [MK1] because it satisfies the above requirements and, also, because there are XPL compilers available for IBM/360 and CDC/6400. Above all, the XPL compiler and SLR(1) analyzer, written in the XPL

CHAPTER 3

A COMPILER WRITING SYSTEM

Compiler writing systems are very useful tools for implementing compilers. Such systems reduce the implementation cost of a compiler; therefore, one can implement more compilers with less effort. They are also excellent tools for a language designer. They encourage experimentation in language design, to determine the effects of various design decisions, especially as they permit one to see the results in a fairly short time. There exist several good compiler writing systems, as discussed in chapter 2. This thesis addresses some specific aspects of compiler writing systems, as follows:

- Error Repair
- Paragraphing
- Semantics

For a compiler writing system we need an implementation language which is readable, has a powerful structuring mechanism and has a compiler which generates efficient code. For this purpose we chose the XPL language [MK1] because it satisfies the above requirements and, also, because there are XPL compilers available for IBM/360 and CDC/6400. Above all, the XPL compiler and SLR(1) analyzer, written in the XPL language, were available at Concordia University.[7] All programs described in this thesis are written in the XPL language.

3.1 SYNTACTIC ANALYZER

Parsing, or syntax analysis, is a process in which the string of tokens (symbols), received from the lexical analyzer, is examined to determine whether the string obeys certain structural conventions explicit in the syntactic definition of the language [AH1]. The parser, when presented with an input string, attempts to construct a parse tree whose leaves match the input string. If it is successful, then the input string is syntactically correct for the given grammar; otherwise, the input string is syntactically incorrect.

There exist bottom-up parsers for LR(K) grammars, where L stands for left to right scan, and R stands for right sentential form. These parsers analyse input strings from left to right, and they attempt to construct the parse tree for a given input string starting from the leaves of the tree to its root; this is called bottom-up parsing.

An LR parser that only reads the current symbol from the input string and does not look ahead at the next symbol to decide which parsing action to take is called an LR(0) parser. The LR parser that looks ahead the next input symbol to decide which parsing action to take is referred as an LR(1) parser.

In general, an LR parser that looks ahead at the next K input symbols to decide which parsing action to take is called an LR(K) parser, where K >= 0.

A smaller subset of LR(K) grammars, which is a large and useful set of grammars, is used for practical purposes. This subset of LR(K) grammars is called SIMPLE LR(K) or SLR(K) grammars [DR1] and has the following properties:

- Adding look-ahead sets to the parsing tables is sufficient to render them useful for analyzing the syntactic correctness of an input string and
- The computation of look-ahead sets is simple

The algorithms for mechanical generation of efficient LR and SLR parsers for context-free grammars are given in [AH2]. The SLR(1) analyzer analyzes a context-free grammar. If the grammar is SLR(K), for 0 <= K <= 1, then it generates the parsing tables and look-ahead sets for it. The generated parsing tables represent the deterministic FSM (finite state machine) for the grammar and are composed of the following:

- Parsing action table
- Parsing go-to table

A brief description of the function of these tables is given below:

- PARSING ACTION TABLE:

Given the current state of the FSM and the incoming input symbol, the parsing action table dictates what action to take. The actions are as follows:

- READ

  Read the current symbol from the input string.

- REDUCE

  A production rule is being recognized; reduce the production.

- LOOK-AHEAD

  A look-ahead to next symbol in the input string is necessary to make any parsing decision.

- ACCEPT

  The given input string is syntactically correct; therefore, it is accepted and parsing is terminated.

- ERROR

  The given input string is syntactically incorrect and error repair is necessary.

- PARSING GO-TO TABLE:

Given the current state of the FSM and the input symbol, the parsing go-to table gives a state number which becomes the current state.

- PARSING ACTION TABLE:

Given the current state of the FSM and the incoming input symbol, the parsing action table dictates what action to take. The actions are as follows:

- READ

  Read the current symbol from the input string.

- REDUCE

  A production rule is being recognized; reduce the production.

- LOOK-AHEAD

  A look-ahead to next symbol in the input string is necessary to make any parsing decision.

- ACCEPT

  The given input string is syntactically correct; therefore, it is accepted and parsing is terminated.

- ERROR

  The given input string is syntactically incorrect and error repair is necessary.

- PARSING GO-TO TABLE:

Given the current state of the FSM and the input symbol, the parsing go-to table gives a state number which becomes the current state.

The states of the deterministic FSM are stored in a push-down stack, since only the current state is ever used at any stage in the parsing process. In 'READ' action we stack the new state. In 'REDUCE' action we replace a string of states from the top of the stack by the new state.

In case of a grammar G which is not LR(0), G has at least one situation where the parser will encounter a read-reduce conflict.

Example: Consider the following production rules for some grammar G.

```
(N1)  G ::= E _!_
(N2)  E ::= E + T
(N3)  E ::= T
(N4)  T ::= T * I
(N5)  T ::= I
```

Fig. 3.1: Grammar G

Where N1 to N5 are production numbers and _!_ denotes the end-of-program symbol. These production rules can be represented by the FSM as shown in Fig. 3.2.

In state n7, when the parser reads 'T', the FSM goes from state n7 to state n8, as shown in Fig. 3.2. At this point the question is what should the parser do now. Should the parser perform the action 'REDUCE' or the action 'READ'. Therefore, state n8 is called an inadequate state. The state n4 is also an inadequate state for the same reason.

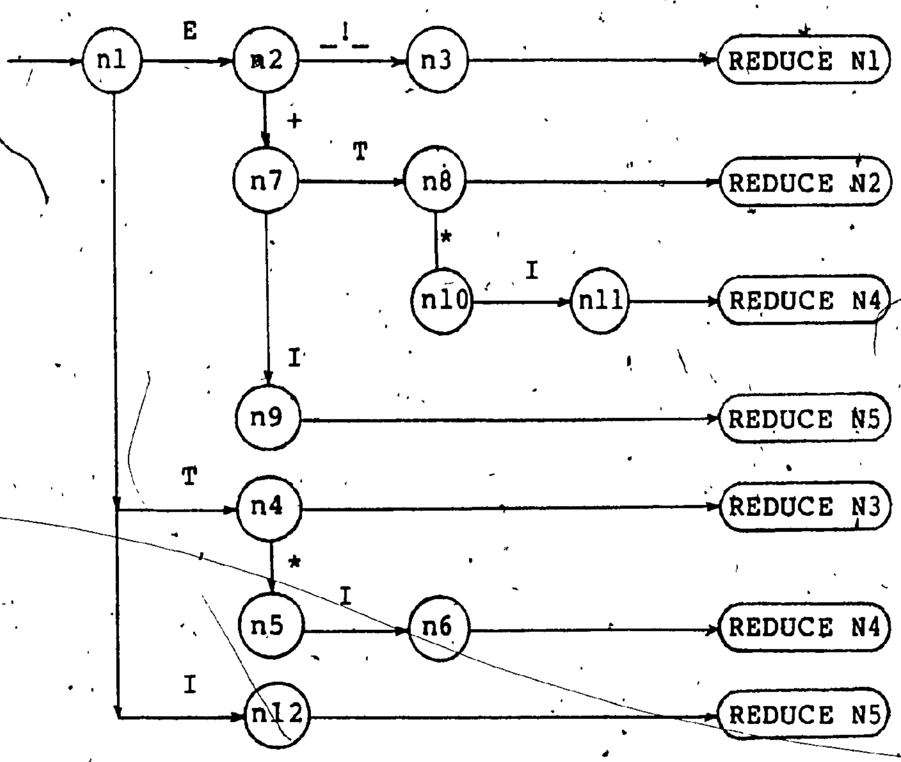


Fig. 3.2: FSM of grammar G.

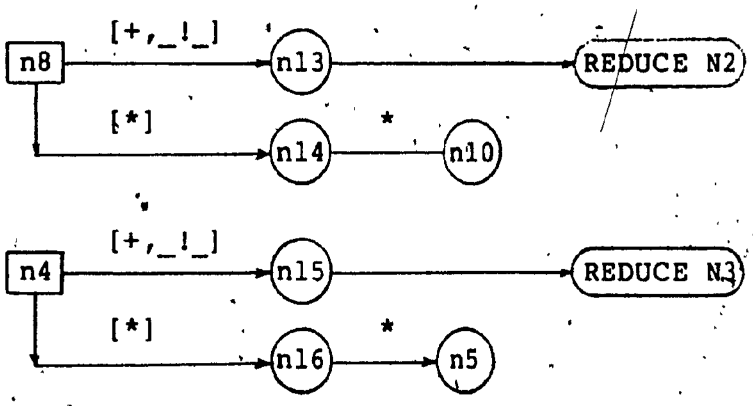


Fig. 3.3: Changes required to convert the FSM of Fig. 3.2 into a Deteministic FSM.

Where n denotes the state of the FSM,
N denotes the production number,
□ indicates the look-ahead state, and
[ ] indicates the look-ahead set.

This situation, however, can be resolved by looking ahead at the point where the FSM goes into an inadequate state, i.e., the FSM enters into a state which is now called a look-ahead state, where the FSM does not try to read; rather, it looks ahead at the symbol to be read and decides accordingly what to do. The changes required to convert the FSM of Fig. 3.2 into a deterministic FSM are shown in Fig. 3.3.

The state n8 in Fig. 3.3 is a look-ahead state. At this point the FSM looks ahead to the next symbol from the input string and determines if the look-ahead symbol is a member of the set [+,_!_] or is a member of the set [*]. The FSM goes to state n13 if the look-ahead symbol belongs to the set [+,_!_]; otherwise (if the look-ahead symbol belongs to the set [*]), it goes to the state n14.

All context-free LR(K) grammars which can be parsed using an FSM and some look-ahead sets associated with the inadequate states are called LALR(K) grammars, and the following condition holds:

$$\mathrm{SLR}(K) \subset \mathrm{LALR}(K) \subset \mathrm{LR}(K)$$

The computation of look-ahead sets for some of the LALR(K) grammars is non-trivial and the complexity of the parsing technique grows with the complexity of the grammar. On the other hand, the computation of look-ahead sets for SLR(K) grammars is simpler and the sets are disjoint.

In practice, the SLR(1) syntax analysis method is a useful and versatile technique for parsing context-free languages in compiling applications. We can picture an SLR(1) parser as shown in Fig. 3.4.

Input symbols are stacked until the parser can perform a reduction (or apply a production rule) and, therefore, the contents of the parse stack change dynamically while the parser is parsing the input program. The parser may make a reduction which may turn out to be incorrect. This may be handled in one of two different ways. The first method is to backup (or to backtrack) to the point where another alternative may be tried. This involves restoring parts of the string to the previous form or erasing some of the reductions made. The second method is to carry out all possible parses in parallel. As some of them lead to 'dead ends' (no more reductions are possible), they are dropped. Either method makes it a little difficult to recover when the parser goes into an error state. In this thesis we have suggested a different and simpler approach for error repair which also avoids the parser going into an error state. This is discussed in section 3.3.

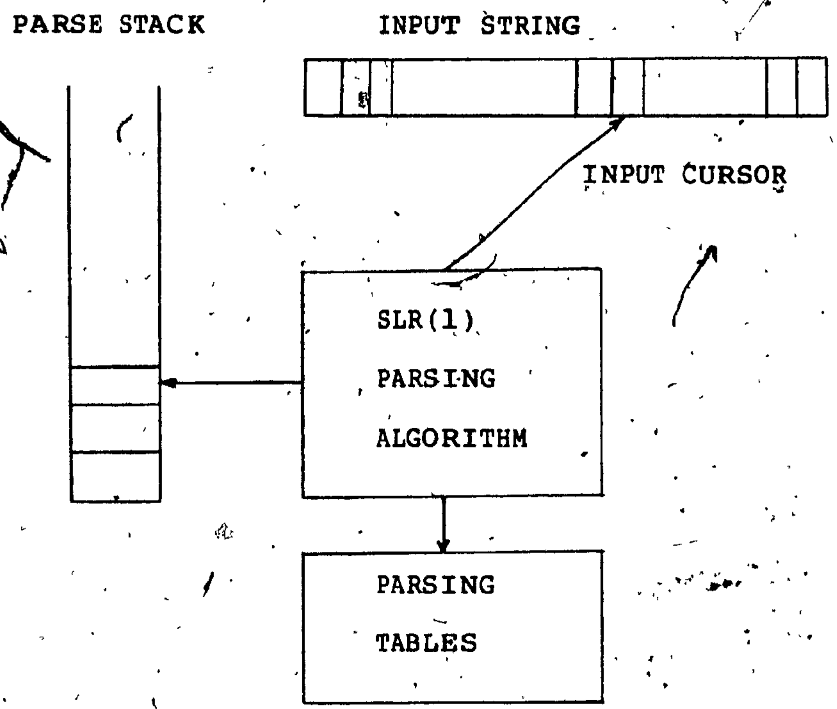


Fig. 3.4: SLR(1) Parser

3.2 SEMANTIC SPECIFICATION

Ideally a compiler writing system should automatically generate the whole compiler. A general notation for semantic specification would permit the development of a true compiler generator, just as BNF led to the development of parser generators. There exist a few compiler writing systems that produce whole compilers [GO1, LE1]. The basic aim of a compiler writing system is to generate as much of a compiler as possible.

When the parser recognizes a grammatical rule, it calls the semantic analyzer or synthesizer. The synthesizer is composed of synthesis procedures and semantic routines which perform all the semantic checking and generate the target language (pseudo or machine language). The standard semantic routines are the static part of the synthesizer and are called from the synthesis procedures which change from language to language.

To generate the synthesis procedures, we associate the semantic specification with each syntactic rule of a grammar, which is called the augmented grammar. The semantics can be specified by the method used in SLAP [GO1] or by Hierarchical Directed Graphs (HDG's) [RU1].

We have selected to specify semantics using the HDG representation. HDG's are a method of representing a program

3.2 SEMANTIC SPECIFICATION

Ideally a compiler writing system should automatically generate the whole compiler. A general notation for semantic specification would permit the development of a true compiler generator, just as BNF led to the development of parser generators. There exist a few compiler writing systems that produce whole compilers [GO1, LE1]. The basic aim of a compiler writing system is to generate as much of a compiler as possible.

When the parser recognizes a grammatical rule, it calls the semantic analyzer or synthesizer. The synthesizer is composed of synthesis procedures and semantic routines which perform all the semantic checking and generate the target language (pseudo or machine language). The standard semantic routines are the static part of the synthesizer and are called from the synthesis procedures which change from language to language.

To generate the synthesis procedures, we associate the semantic specification with each syntactic rule of a grammar, which is called the augmented grammar. The semantics can be specified by the method used in SLAP [GO1] or by Hierarchical Directed Graphs (HDG's) [RU1].

We have selected to specify semantics using the HDG representation. HDG's are a method of representing a program in which the control flow is expressed in the form of a flowchart notation. The HDG representation is convenient for expressing the program, and also facilitates effective program optimization.

3.2.1 A REVIEW OF HIERARCHICAL DIRECTED GRAPHS (HDG'S)

Rudmik proposed to represent a program by HDG's [RU1]. The HDG is constructed from Control Structure Graphs (CSG's). CSG's consist of four basic nodes, each of which is proper, reducible and constructed from the basic structure nodes as given in Fig. 3.5. The properties of these nodes are used to determine the control flow of the CSG. No other type of node is permitted except for a single node having no entry edge and no exit edge. This node represents the program and contains the rest of the HDG.

Fig. 3.8 illustrates an HDG representation of a sample XPL program as given in Fig. 3.7, where exp1 and exp2 are arithmetic expressions and Si (i=1,...,n) are statements. Fig. 3.6 illustrates the set of CSG's used. A CSG being contained within a particular node is indicated by a dotted directed edge from that node to the CSG contained in that node.

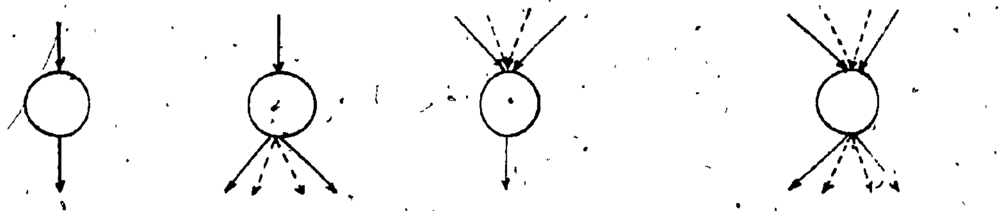


FIG. 3.5: BASIC STRUCTURE NODE

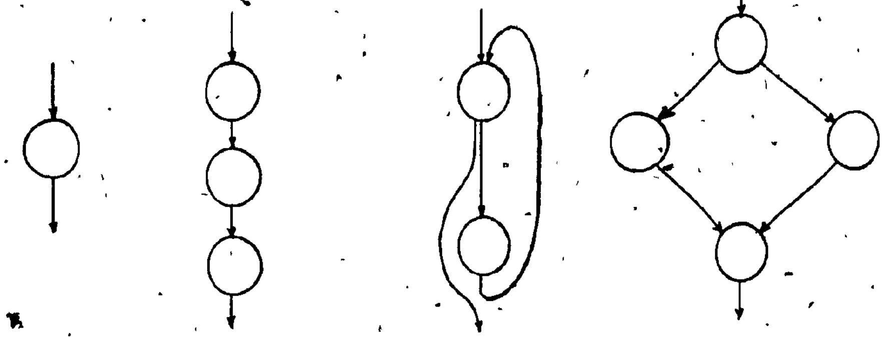


FIG. 3.6: SET OF CSG'S

```
DO WHILE (expl);
    S1;
    S2;
    IF (exp2) THEN
       S3;
    ELSE
       S4;
    S5;
END;
```

FIG. 3.7: A PROGRAM

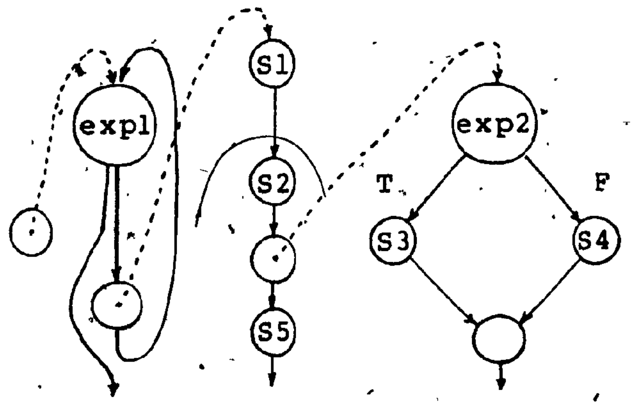


FIG. 3.8: HDG REPRESENTATION OF PROGRAM OF FIG. 3.7

3.2.2 EXAMPLE OF SEMANTIC SPECIFICATION

Let us consider the following syntax rule:

<ARITHMETIC EXP> ::= <ARITHMETIC EXP> + <TERM>

By the time that the parser recognizes this rule, it has already built the partial derivation trees for <ARITHMETIC EXP> and <TERM>. The semantic meaning we want to associate with the above-mentioned production rule is to join the two partial trees with a node having the arithmetic operator '+', and this can be specified in HDG notation as follows:

```
G1(MP) = BINARY_EXPRESSION(OP_ADD,G1(MP),G1(SP));
```

where MP and SP are pointers into the parse stack G1 which point to the symbols on the extreme left and extreme right of the right hand side of the production rule. G1(MP) and G1(SP) point to the partially built trees for the symbols pointed at by MP and SP. 'BINARY_EXPRESSION' is a function which performs the above-mentioned operation. It is shown in Fig. 3.9(a) and in Fig. 3.9(b). Fig. 3.9(a) represent the state of the parse stack, for the right hand side of the above-mentioned production rule, before the function 'BINARY_EXPRESSION' is called. Fig. 3.9(b) represent the state of the parse stack, for the left-hand side of the said production rule, after the function 'BINARY_EXPRESSION' is called.

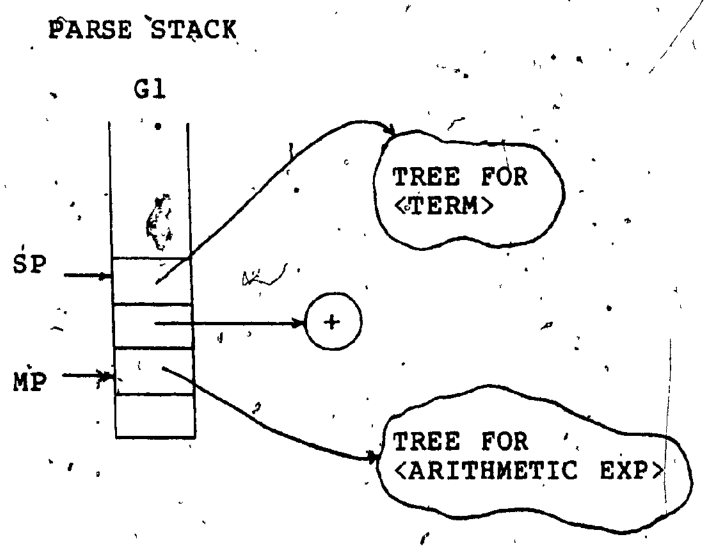


FIG. 3.9(a): STATE OF PARSE STACK BEFORE CALLING THE FUNCTION 'BINARY_EXPRESSION'.

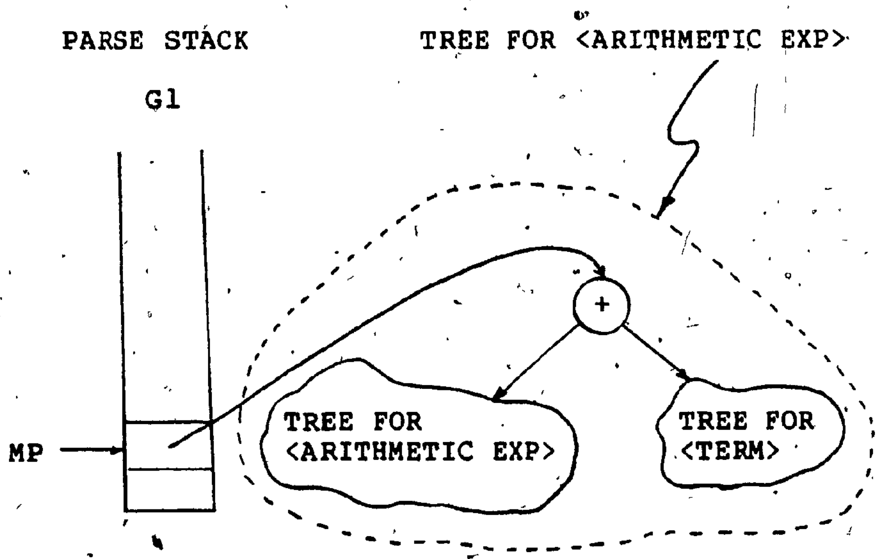


FIG. 3.9(b): STATE OF PARSE STACK AFTER CALLING THE FUNCTION 'BINARY_EXPRESSION'.

In the grammar we will associate the semantics with the syntax of the above-mentioned production rule as follows:

```
<ARITHMETIC EXP> ::= <ARITHMETIC EXP> + <TERM>
G1(MP) = BINARY_EXPRESSION(OP_ADD,G1(MP),G1(SP));
```

This is called the augmented grammar, which will be input to the language analyzer in order to generate the synthesis procedures. A complete example is given in Appendix A.

3.3 ERROR RECOVERY

Many programs that are submitted to compilers will have errors. These errors can be classified as follows:

- compiler errors
- semantic errors
- syntax errors

Compiler errors are caused by faulty design or by implementation-defined limitations of a compiler. Semantic errors indicate that the program submitted to the compiler violates the semantic specification of the programming language. Finally, syntax errors indicate that the program is not contructed according to the syntax rules of the language.

In an educational environment most compilation results in compile-time errors. Many programs submitted to the compiler

In the grammar we will associate the semantics with the syntax of the above-mentioned production rule as follows:

```
<ARITHMETIC EXP> ::= <ARITHMETIC EXP> + <TERM>
G1(MP) = BINARY_EXPRESSION(OP_ADD,G1(MP),G1(SP));
```

This is called the augmented grammar, which will be input to the language analyzer in order to generate the synthesis procedures. A complete example is given in Appendix A.

3.3 ERROR RECOVERY

Many programs that are submitted to compilers will have errors. These errors can be classified as follows:

- compiler errors
- semantic errors
- syntax errors

Compiler errors are caused by faulty design or by implementation-defined limitations of a compiler. Semantic errors indicate that the program submitted to the compiler violates the semantic specification of the programming language. Finally, syntax errors indicate that the program is not contructed according to the syntax rules of the language.

In an educational environment most compilation results in compile-time errors. Many programs submitted to the compiler have more syntax errors than any other kind of error. Our main interest is to detect and repair syntax errors in order to permit the compilation process to continue. Different compilers react in different ways to syntax errors. Our error repair technique repairs the syntactically incorrect sentence to a valid one, and consequently helps the compiler to continue parsing. We use the term 'repair' to suggest that the modified sentence is not necessarily what the programmer intended. Error repair permits compilation to continue with three possible outcomes:

1) Correctly repair the error to what the programmer had intended, e.g.,

   - Incorrect code:
     ```
     A = B + C * D
     Z = Z * C;
     ```
   - Corrected code:
     ```
     A = B + C * D;
     Z = Z * C;
     ```

2) Repair the syntax of the program but change the intended meaning, e.g.,

- Incorrect code:

```
A = (B + C*D/E;
```

- Repaired code:

```
A = (B + C*D/E);
```

- Programmer's intention could be one of the following:

```
(1)  A = (B + C)*D/E;
(2)  A = (B + C*D)/E;
(3)  A = (B + C*D/E);
```

3) Repair the syntax of the program but introduce subsequent errors, e.g.,

- Incorrect code:

```
A = B + C  D
Z = Z*C;
```

- Repaired code:

```
A = B + C; D;
Z = Z*C;
```

- Intended code:

```
A = B + C*D;
Z = Z*C;
```

3.3.1 MOTIVATION FOR ERROR REPAIR

Since it is clearly difficult for a compiler to handle errors in any non-trivial way, the aim is to develop a better error repair technique in order to allow the compilation process to continue. In fact, many compilers that are widely used do not have good error recovery facilities, although there are a few exceptions (e.g., PL/C [CO2], SP/k [BA1]). This seems to be a consequence of defining the purpose of a compiler in a rather narrow sense as something which translates 'syntactically and semantically valid programs'. Given this goal, it is conceivably acceptable to have error conditions cause printing of a message and termination of compilation. However, we would define compilers as tools to help humans in the creation of programs. The tool should help the user as much as possible, while not introducing any errors of its own. As Knuth [KN2] says:

> "Another important way for a production program to be good is for it to interact gracefully with its users, especially when recovering from human errors in the input data."

The error repair technique in this thesis is designed to be used in an educational environment. Syntax repair makes a modification and the parser then continues processing the program so that errors do not inhibit further checking.

Through the use of syntax repair, we hope to cut down the effort required to get a program working correctly. It is expected that this will result in a more efficient use of computer resources and supplies, because fewer runs should be required to complete an assignment. Conway and Wilcox [CO2] point out that observation by users of different installations using PL/C indicates that 30 to 50 per cent fewer runs are needed on a syntax-repairing compiler.

A good error repair technique in a compiler can make it a better programming tool if it can perform the following:

- Print out error messages indicating the nature and location of the error.
- Indicate how the errors were repaired.
- Print out the repaired source statement.

According to Conway and Wilcox [CO2]:

"...most effective communication rests not in the specific messages but in a reconstruction of the source statement that syntactic analysis is actually passing on to semantic analysis."

For example, given the input sentence

```
    C = ((A*(B + A)/(A;
```

we would display the repaired line

```
    C = ((A*(B + A)/(A)));
```

Some compilers print the error messages following the source statement having syntax error(s), while others print them at the end of the compiled listing. A compiler generated by the proposed compiler writing system prints the repaired source statements as the compilation listing. After the compilation listing is a listing of the error messages, if there are any. Since the proposed compiler writing system also includes paragraphing, there is no one-to-one correspondence between line numbers of the input text and line numbers of the compilation listing (paragraphed output). Therefore, each error message has a card number of the input text and a line number of the paragraphed output, indicating the location of the error. These are followed by, on the same line, a message specifying the nature of the error. The second line of an error message indicates how the error was repaired. An example of the error messages produced by a generated compiler is given in Appendix B.

3.3.2 SYMBOL PAIR MATCHING ERROR REPAIR

In order to recover from a certain class of errors, it is desirable to have a history of the parser steps. Most bottom-up parsing techniques do not maintain this history.

We have adopted a simple approach for our error repair technique. It seems to be effective, efficient and easy to implement. By this technique the errors are recovered in the scan phase. We believe in recognizing the error as early as possible, i.e., not waiting until the parser recognizes the error in the input text, when it can be recognized by the scanner. This approach has the advantage that the proposed error repair technique can be used in conjunction with different parsing techniques. In addition, parser error recovery can be used to augment the error handling capability of our compiler.

The lexical analyzer (scanner) is a translator whose input is the string of symbols representing the source program and whose output is a stream of tokens (symbols). What constitutes a token is implied by the specification of the programming language. These tokens form the input to the syntactic analyzer (parser).

Parsing, or syntactic analysis, is a process in which the string of tokens is examined to determine whether the string obeys certain structural conventions explicit in the syntactic definition of the language [AH1].

The parsers for most programming languages require looking ahead one token (symbol) to resolve any read-reduce conflicts, as discussed in section 3.1. Therefore, most of the parsers always have two tokens or symbols from the input program at any given time to work with. In this thesis these tokens or symbols will be referred to as:

- 'current_symbol'
- 'next_symbol'

The parser normally works with 'current_symbol' unless it is required to look ahead; only then does it look at 'next_symbol' to make the decision. When the parser needs another symbol from the input program, it first copies 'next_symbol' into 'current_symbol', and then invokes the lexical analyzer to get a new token as 'next_symbol'.

'current_symbol' and 'next_symbol' make a pair. In the proposed error repair technique, the validity of the pair of symbols ('current_symbol' and 'next_symbol') is checked through a boolean matrix which is referred to as the error table. The error table only specifies whether 'current_symbol' can be followed by 'next_symbol', for any given pair of symbols. The automatic generation of the error table will be discussed in section 3.3.3.

definition of the language [AH1].

The parsers for most programming languages require looking ahead one token (symbol) to resolve any read-reduce conflicts, as discussed in section 3.1. Therefore, most of the parsers always have two tokens or symbols from the input program at any given time to work with. In this thesis these tokens or symbols will be referred to as:

- 'current_symbol'
- 'next_symbol'

The parser normally works with 'current_symbol' unless it is required to look ahead; only then does it look at 'next_symbol' to make the decision. When the parser needs another symbol from the input program, it first copies 'next_symbol' into 'current_symbol', and then invokes the lexical analyzer to get a new token as 'next_symbol'.

'current_symbol' and 'next_symbol' make a pair. In the proposed error repair technique, the validity of the pair of symbols ('current_symbol' and 'next_symbol') is checked through a boolean matrix which is referred to as the error table. The error table only specifies whether 'current_symbol' can be followed by 'next_symbol', for any given pair of symbols. The automatic generation of the error table will be discussed in section 3.3.3.

Whenever the parser needs a new token, it invokes the lexical analyzer to get one. Consequently, the parser has a new pair of symbols. Before the parser uses this new pair of symbols, the error repair process takes over control from the parser. First the validity of the pair of symbols is checked through the error table. If the pair is syntactically correct, then control is given back to the parser. Otherwise, before control is given back to the parser, one of the following actions (specified by compiler writer or language designer) is taken (by default the action is to skip the next symbol):

- SKIP NEXT_SYMBOL

  Example:

  ```
    A = B+*C;
  ```

  When 'current_symbol' is '+' and 'next_symbol' is '*' then '*' will be skipped and repaired statement will be as follows:

  ```
    A = B+C;
  ```

Whenever the parser needs a new token, it invokes the lexical analyzer to get one. Consequently, the parser has a new pair of symbols. Before the parser uses this new pair of symbols, the error repair process takes over control from the parser. First the validity of the pair of symbols is checked through the error table. If the pair is syntactically correct, then control is given back to the parser. Otherwise, before control is given back to the parser, one of the following actions (specified by compiler writer or language designer) is taken (by default the action is to skip the next symbol):

- SKIP NEXT_SYMBOL

  Example:

  ```
  A = B+*C;
  ```

  When 'current_symbol' is '+' and 'next_symbol' is '*' then '*' will be skipped and repaired statement will be as follows:

  ```
  A = B+C;
  ```

- REPLACE CURRENT_SYMBOL BY A TERMINAL SYMBOL

Example:

```
RANDOM_NO;
   PROCEDURE(I);
      .

      .

   END RANDOM_NO;
```

When 'current_symbol' is ';' and 'next_symbol' is 'PROCEDURE' then ';' will be replaced by ':' and the repaired text will be as follows:

```
RANDOM_NO:
   PROCEDURE(I);
      .

      .

   END RANDOM_NO;
```

- REPLACE NEXT_SYMBOL BY A TERMINAL SYMBOL

Example:

```
DECLARE I FIXED:
```

When 'current_symbol' is ~~'FIXED' and~~ 'next_symbol' is ':' then ':' will be replaced by ';' and the repaired text will be as follows:

```
DECLARE I FIXED;
```

- REPLACE CURRENT_SYMBOL BY A TERMINAL SYMBOL

Example:

```
RANDOM_NO;
    PROCEDURE(I);
       .
       .
    END RANDOM_NO;
```

When 'current_symbol' is ';' and 'next_symbol' is 'PROCEDURE' then ';' will be replaced by ':' and the repaired text will be as follows:

```
RANDOM_NO:
    PROCEDURE(I);
       .
       .
    END RANDOM_NO;
```

- REPLACE NEXT_SYMBOL BY A TERMINAL SYMBOL

Example:

```
DECLARE I FIXED:
```

When 'current_symbol' is ~~'FIXED' and~~ 'next_symbol' is ':' then ':' will be replaced by ';' and the repaired text will be as follows:

```
DECLARE I FIXED;
```

- INSERT A TERMINAL SYMBOL BETWEEN CURRENT_SYMBOL AND NEXT_SYMBOL

  Example:

  ```
  A = B+C
  Z = Z*C;
  ```

  When 'current_symbol' is an identifier 'C' and 'next_symbol' is also an identifier 'Z' then ';' will be inserted between the two identifiers and the repaired text will be as follows:

  ```
  A = B+C;
  Z = Z*C;
  ```

In addition to the above-mentioned error repairs, this method also recovers from errors which are due to missing terminal symbols which should occur in pairs. In this thesis we refer to these symbols as grouping elements. For example, the most common terminal symbol pairs which fall in this category are:

(	)
DO	END
PROCEDURE	END

- INSERT A TERMINAL SYMBOL BETWEEN CURRENT_SYMBOL AND NEXT_SYMBOL

  Example:

  ```
  A = B+C
  Z = Z*C;
  ```

  When 'current_symbol' is an identifier 'C' and 'next_symbol' is also an identifier 'Z' then ';' will be inserted between the two identifiers and the repaired text will be as follows:

  ```
  A = B+C;
  Z = Z*C;
  ```

In addition to the above-mentioned error repairs, this method also recovers from errors which are due to missing terminal symbols which should occur in pairs. In this thesis we refer to these symbols as grouping elements. For example, the most common terminal symbol pairs which fall in this category are:

(	)
DO	END
PROCEDURE	END

Consider the following input statements for parenthesis checking:

```
1) DECLARE(VAR1,VAR2 FIXED;
2) DECLARE VAR3(10 FIXED INITIAL(0,1,2,3,4,5,6,7,8,9,10;
3) C(I = ((A*(B+A)/(A;
4) IF(I>J THEN OUTPUT='TRUE';
```

These statements will be repaired as:

```
1) DECLARE (VAR1,
              VAR2) FIXED;
2) DECLARE VAR3(10) FIXED INITIAL(
       0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
3) C(I) = ((A*(B + A)/(A)));
4) IF (I > J) THEN
       OUTPUT = 'TRUE';
```

The proposed error repair technique localizes this type of error by keeping check-points to permit efficient repair. In the above example, when the scanner recognizes the symbol 'FIXED', 'THEN', ';', or '=', the error repair method checks at these points whether all the left parentheses are matched with corresponding right parentheses. If right parentheses are missing, then they are supplied. Also at any given point, if the scanner recognizes a right parenthesis, which also marks a check-point, then it is checked whether there exists a left parenthesis with which it can be matched (details of this check are given in section 3.3.4 under procedure 'check_group_elements'). If so, then the right parenthesis is accepted; otherwise, it is rejected as an input symbol.

The check-points for grouping elements, as mentioned above, are determined by analyzing the grammar manually. These check-points help in confining the errors and, consequently, improve the error repair. If all the grouping element pairs are disjoint, then they provide better check-points. For example, the pair [(,)] is disjoint from other pairs, while [PROCEDURE, END] and [DO, END] are not. In this case the only check point we can have for these pairs is the end of file (EOF) mark of the input text. Only at this point is it possible to determine if all the DO's and PROCEDURE's have their matching END's. Therefore, we have changed the XPL grammar so that the PROCEDURE is terminated by END_PROC. Now the grouping pairs [(,)], [DO, END], and [PROCEDURE, END_PROC] are all disjoint. When the scanner recognizes END_PROC, it is checked if all the DO's have their matching END's, thus confining the error. Also at the EOF mark, it is checked whether all the DO's and the PROCEDURE's have their matching END's and END_PROC's, respectively. Consider the following input text for grouping element checking.

INPUT TEXT:

```
FLUSH_CARDS:
   PROCEDURE;  /* FLUSHES THE CARDS */
      DO WHILE LENGTH(BUFFER) NE 0;
         BUFFER = INPUT;
         CARDCOUNT = CARDCOUNT + 1;
         IF LENGTH(BUFFER) > 0 THEN
            DO;
               OUTPUT = BUFFER;
   END_PROC FLUSH_CARDS;
```

REPAIRED TEXT:

```
FLUSH_CARDS:
   PROCEDURE;  /* FLUSHES THE CARDS */
      DO WHILE LENGTH(BUFFER) NE 0;
         BUFFER = INPUT;
         CARDCOUNT = CARDCOUNT + 1;
         IF LENGTH(BUFFER) > 0 THEN
            DO;
               OUTPUT = BUFFER;
            END;
      END;
   END_PROC FLUSH_CARDS;
```

Further discussion on grouping elements and their check-points for a given grammar is given in Appendix A. More examples of error repair are given in Appendix B.

3.3.3 ALGORITHM FOR GENERATION OF ERROR_TABLE

As was mentioned in section 3.3.2, the 'error_table' for a grammar is a boolean matrix which specifies for any given pair of symbols (i.e., 'current_symbol' and 'next_symbol') whether the current symbol can be followed by the next symbol. An element of the 'error_table' can have the value one or zero for a given pair of symbols. If the value is one, then the symbol pair is valid, and if the value is zero then the symbol pair is invalid.

A very simple method is used for automatic generation of the matrix 'error_table'. This method analyzes the characterstic finite state machine generated by the SLR(1) language analyzer. The following is the algorithm which is used to generate the 'error_table'.

1) Each element of 'error_table' is set to false.

2) Each read state is inspected and for all terminal symbols (s2) which may legally follow (legal transitions) the accessing terminal symbol (terminal symbol before read state) s1, 'error_table(s1, s2)' is set to true.

3) Each production of the grammar which terminates in a terminal symbol (e.g., s1) is inspected. For each symbol s2 in the look-ahead set of that production head, 'error_table(s1, s2)' is set to true.

3.3.4 ALGORITHM FOR PROCEDURE ERROR_REPAIR

A simplified block diagram of data flow for the error repair technique is shown in Fig. 3.10. Here we will briefly discuss the functions of each block.

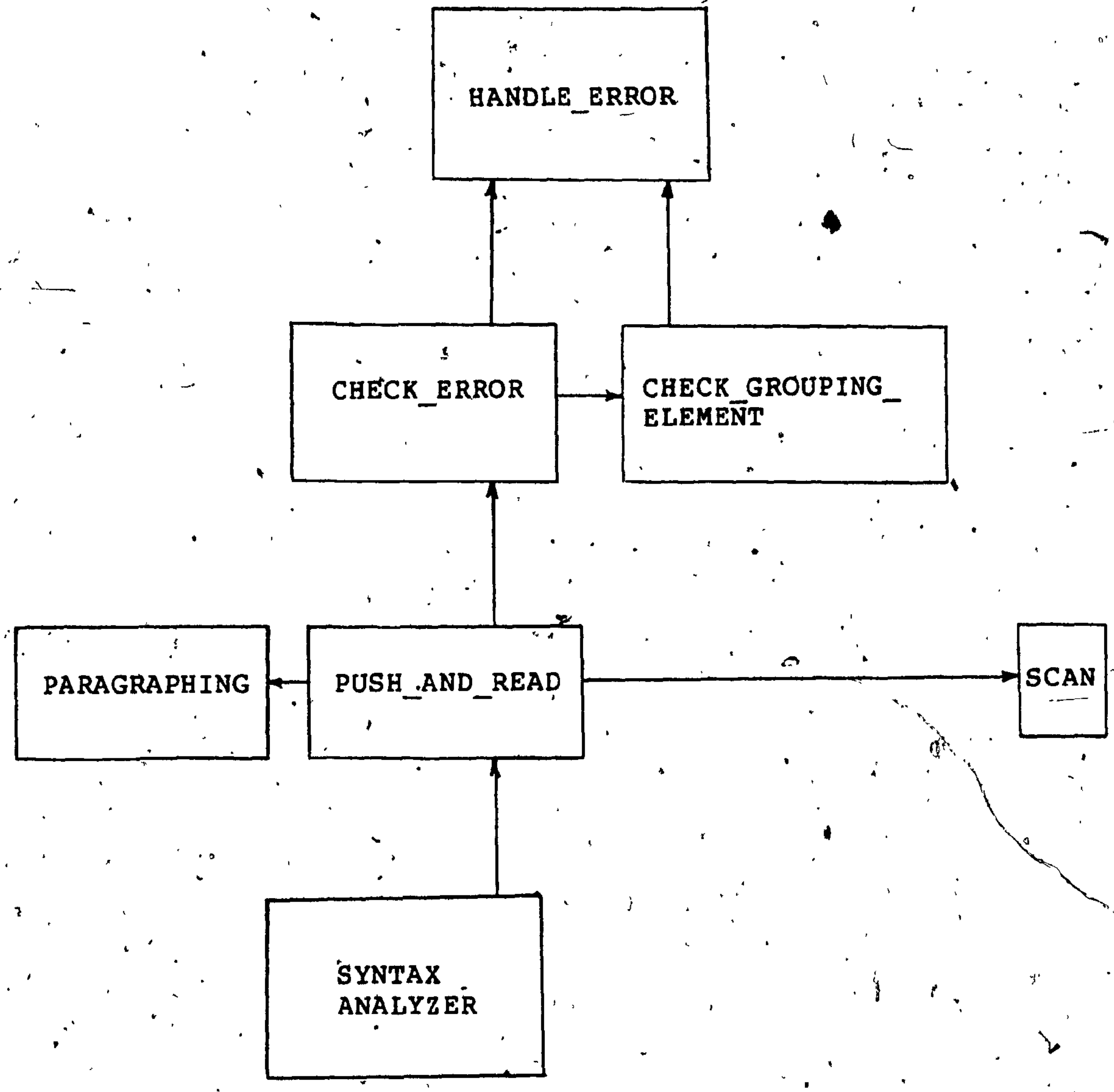
When the syntax analyzer needs the next symbol, it calls the procedure 'push_and_read'. The nucleus of this procedure, which plays the most important part in error repair, is as follows:

```
PUSH_AND_READ:
   PROCEDURE;
      CURRENT_SYMBOL = NEXT_SYMBOL;
      ACTION = 0;
      DO WHILE ACTION = 0;
         IF ERROR_DETECTED THEN
            DO;
               NEXT_SYMBOL = NEW_SYMBOL;
               ERROR_DETECTED = FALSE;
            END;
         ELSE
            DO;
               CALL SCAN;
            END;
         CALL CHECK_ERROR;
      END;
      CALL PARAGRAPHING;
   END PUSH_AND_READ;
```

The flag 'error_detected' and the variable 'new_symbol' are set in the procedure 'handle_error' which will be discussed later. The variable 'action' is set in the procedure 'check_error', and its value is zero if the action to take is to skip the 'next_symbol'. Thus, when this procedure is called, it first makes 'next_symbol' to be the 'current_symbol' and sets 'action' to zero; then, it goes into a loop as long as 'action' remains zero. Inside the loop it checks for the flag 'error_detected'. If it is false, then the scanning procedure is called to get the next symbol. But, if the flag is true, then the next symbol is received from the temporary storage called 'new_symbol'. By now we have a new pair of symbols, i.e., 'current_symbol' and 'next_symbol', so the procedure 'check_error' is called. If this procedure sets the value of 'action' to zero, then the implication is that the next symbol has to be skipped. We will therefore remain in the loop. For any value of 'action' other than zero, procedure 'paragraphing' is called.

```
      DO WHILE ACTION = 0;
         IF ERROR_DETECTED THEN
            DO;
               NEXT_SYMBOL = NEW_SYMBOL;
               ERROR_DETECTED = FALSE;
            END;
         ELSE
            DO;
               CALL SCAN;
            END;
         CALL CHECK_ERROR;
      END;
      CALL PARAGRAPHING;
   END PUSH_AND_READ;
```

The flag 'error_detected' and the variable 'new_symbol' are set in the procedure 'handle_error' which will be discussed later. The variable 'action' is set in the procedure 'check_error', and its value is zero if the action to take is to skip the 'next_symbol'. Thus, when this procedure is called, it first makes 'next_symbol' to be the 'current_symbol' and sets 'action' to zero; then, it goes into a loop as long as 'action' remains zero. Inside the loop it checks for the flag 'error_detected'. If it is false, then the scanning procedure is called to get the next symbol. But, if the flag is true, then the next symbol is received from the temporary storage called 'new_symbol'. By now we have a new pair of symbols, i.e., 'current_symbol' and 'next_symbol', so the procedure 'check_error' is called. If this procedure sets the value of 'action' to zero, then the implication is that the next symbol has to be skipped. We will therefore remain in the loop. For any value of 'action' other than zero, procedure 'paragraphing' is called.



CONTROL FLOW OF THE ERROR REPAIR TECHNIQUE

FIG. 3.10

CHECK_ERROR:

The 'check_error' procedure has two distinct parts. When this procedure is called, it checks through 'error_table' whether the current pair of symbols is legal or illegal.

If the pair is legal, then 'action' is first set to 1 so that on return to 'push_and_read' we will get out of the loop. Then it is checked whether the next symbol is a member of the grouping elements or their check points. If it is not, then control is returned to the calling procedure; otherwise, the procedure 'check_grouping_element' is called. After return from this procedure, the error flags are checked. If there was no error, then control is simply returned to the calling procedure; otherwise, a proper action, depending on the error, is assigned and procedure 'handle_error' is invoked before returning to the calling procedure.

The other part of the 'check_error' procedure is executed when the current pair of symbols is illegal. In this event, 'action_table' is searched to see if an action is specified by the user (compiler writer or language designer) for this pair. If there is no data in the action table for this pair, then, by default, it is assumed that the next symbol has to be skipped; otherwise, the action is retrieved from the action table and the procedure 'handle_error' is called before returning to the calling procedure.

HANDLE_ERROR:

The handle_error procedure is composed of one DO CASE statement which handles only four cases. These cases are listed below:

CASE(0):

When the next symbol has to be skipped, then this case is executed. It simply prints the proper message. Control is then returned to the calling procedure.

CASE(1):

When a terminal symbol has to be inserted between the current pair of symbols, then this case is executed. In this case, the flag 'error_detected' is set to true and the variable 'new_symbol' is assigned the value of 'next_symbol'. If this procedure was invoked because an illegal pair of symbols was encountered (rather than due to an error being flagged by the 'check_group_elements' procedure), then the symbol to be inserted is taken from 'action_table' and assigned to 'next_symbol'. Now we have a new pair of symbols, so procedure 'check_group_elements' is called. At this point it does not matter who invoked this procedure; the program proceeds by checking if there was any error flagged by procedure 'check_group_elements'. If there was no error, then it simply prints the message about what is done here before returning to the calling procedure. In the event of an error, it is handled first.

CASE(2):

When the current symbol has to be replaced by some other terminal symbol, then this case is executed before control is transfered to the calling procedure. The new value of 'current_symbol' is taken from 'action_table' and a proper message for this replacement is issued. Procedure 'check_group_elements' is invoked next.

CASE(3):

This is the same as case(2), except here we deal with 'next_symbol' instead of 'current_symbol'.

Before this procedure is invoked, the value of 'action' should be known; this would, subsequently, specify the case to be executed. The value of 'action' is either retrieved from 'action_table', or specifically assigned in the program logic.

CHECK_GROUP_ELEMENTS:

The procedure 'check_group_elements' is invoked whenever we have a new pair of symbols, to check if the next symbol is one of the grouping elements or marks one of their check points. This procedure also consists of one DO CASE statement, which deals with a number of cases. The number of cases depends upon the number of grouping elements, and on the number of groups of their check points. Case(0) is reserved and only contains a null statement. It is executed only when the next symbol is not a grouping element or does not mark a check point. Each of the other cases performs a unique function. For example, if we only consider the left and right parentheses, then if 'next_symbol' is a '(', case(1) will be executed. Here we simply increment the counter for left parentheses by one. If 'next_symbol' is ')', case(2) will be executed. Here we first increment the counter for right parenthesis, and then check if we have more right parentheses than left parentheses. If so, there is an error; otherwise, we return to the calling procedure. In the event of an error, we issue an error message, decrement the right parenthesis counter by one and set the flag for right parenthesis error; then we return to the calling procedure.

If 'next_symbol' marks a check point for parentheses, then case(6) will be executed. Here we simply check if the number of left parentheses is greater than the number of right parentheses. If not, then we set their respective counters to zero and return to the calling procedure. If so, we issue an error message for unmatched left parenthesis and set up the proper flag for this error before returning to the calling procedure.

The other case statements follow almost the same philosophy. In 'check_group_elements' we have nine cases in all, which are listed below; the tokens, or their symbolic names, inside the comment delimiters indicate when each case will be executed:

```
     CASE(0):   /* NOT USED                  */
     CASE(1):   /* LEFT PARENTHESIS          */
     CASE(2):   /* RIGHT PARENTHESIS         */
     CASE(3):   /* PROCEDURE                 */
     CASE(4):   /* DO                        */
     CASE(5):   /* END                       */
     CASE(6):   /* ; FIXED CHARACTER THEN */
     CASE(7):   /* END_PROC                  */
     CASE(8):   /* EOF                       */
```

3.4 PARAGRAPHING

The output listing that is produced by the compiler is intended to reflect the logical structure of the program, rather than the physical structure of the input text, and to improve the readability of the output listing. The proposed paragraphing technique can be automated to expose the logical structure of the input program. The technique considered can be used to paragraph program components, such as comments, macros, adding blank lines etc., which are not part of the language. Therefore, it is difficult to automatically generate a paragraphing module having the above-mentioned characterstics for a given language. Instead, we have coded a paragraphing module which is sufficient for most of the XPL-like languages. It may require modification to handle programming structures which are not in XPL-like languages. The modification will be minimal and straightforward. The algorithm for the paragraphing technique is given in section 3.4.3. This section deals with some of the reasons for compiler-controlled paragraphing and discusses the details of our paragraphing technique.

The basic aim of paragraphing is to display the logical structure of the input program and to make it more readable on the printed page. This helps in conveying information about the program and its understanding, especially to someone other than the programmer. In this respect, consistency in the paragraphing style is very important, because it enhances the readability of the programs. Automatic paragraphing is certainly consistent. It is easier to understand and visualize the structure of a program paragraphed in a familiar style (i.e., some preferred style), than it is for a program paragraphed in an unfamiliar style. Paragraphing is more important in a pedagogical environment. When a student always sees his programs paragraphed, it helps him to learn the close relationships between program structure (emphasized by paragraphing), program construction and program execution.

To achieve the above objectives, the proposed paragraphing technique rebuilds the entire input text from the symbols (tokens) produced by the scanner. Each symbol received by the paragraphing module, or the paragrapher, is appended into the output buffer. The output buffer is printed out (emptied) under two different circumstances. Firstly, the program logic of the paragrapher may force the output buffer to be printed out when the paragraphing rules call for the new line. Secondly, due to the limited length of the output buffer (which is kept the same as the limited length of the physical print line), when a symbol cannot be appended into the output buffer then the output buffer, in its present state, is printed out, resulting in the truncation of the source statement. The output buffer is then re-initialized with the left margin indented one level to the right. The symbol is now placed into the output buffer, and the process of paragraphing continues.

Like most parsers, this paragraphing technique also looks ahead one symbol, in order to make certain decisions related to achieving its objective. It was mentioned in the above paragraph, that, when the output buffer is printed out, it is re-initialized, and the left margin is set. There are cases where the next incoming symbol dictates the left margin of the next output line. (For example, consider the compound statement 'DO; {do-body} END;', where the symbol 'DO' will cause the do-body to be indented one level to the right). Before each statement of the do-body is printed out, a look ahead will be performed to see if the next symbol is 'END'. If it is, then the indentation level will be adjusted one level to the left, otherwise the indentation level will remain unchanged. Now the output buffer is printed out and it will be re-initialized with the proper left margin. Detailed examples of paragraphed output are given in Appendix B. In the following sections we will highlight specific aspects of our paragraphing method, with short examples.

3.4.1 PARAGRAPHING TO EXPOSE LOGICAL STRUCTURE

To highlight the logical structure of the input program, three basic constructs of the source language are considered. These are the simple statement, the compound statement, and the procedure, and are discussed below.

3.4.1.1 SIMPLE STATEMENTS

Each simple statement goes on a new line with the current left-hand margin, so that the vertical spacing reflects the logical sequence of statements. If the source statement cannot fit on the output line, then the statement is truncated and the partial statement is printed. The continuation of the statement is printed on the next line(s) with the left margin indented. The following is an example of an input text and its paragraphed output:

INPUT TEXT:

```
A=B+C;D=E*F;G=D/A*(D+A);OUTPUT=G;
```

PARAGRAPHED OUTPUT:

```
A = B + C;
D = E*F;
G = D/A*(D + A);
OUTPUT = G;
```

3.4.1.2 COMPOUND STATEMENTS

The body of a compound statement is indented so that the horizontal spacing reflects logical subordination. All information at one logical level is listed with a common left margin. The compound statements are DO, DO WHILE, DO CASE and IF statements. The following is an example of an input text and its paragraphed output:

INPUT TEXT:

```
J=0;DO I=1 TO 10;J=J+1;END;I=0;DO WHILE(J>0);
J=J-1;I=I+1;END;IF I>10 THEN OUTPUT=
"I IS GREATER THAN 10.";
ELSE OUTPUT="I IS LESS THAN OR EQUAL TO 10.";
OUTPUT="END OF TEST";
```

PARAGRAPHED OUTPUT:

```
J = 0;
DO I = 1 TO 10;
   J = J + 1;
END;
I = 0;
DO WHILE (J > 0);
   J = J - 1;
   I = I + 1;
END;
IF I > 10 THEN
   OUTPUT = "I IS GREATER THAN 10.";
ELSE
   OUTPUT = "I IS LESS THAN OR EQUAL TO 10.";
OUTPUT = "END OF TEST.";
```

3.4.1.3 PROCEDURES

The procedures are paragraphed in a style which is preferable to many programmers, that is, the name of the procedure is printed on a new line with the current left margin. The keyword 'PROCEDURE' followed by the argument list, if any, goes on a new line indented one level to the right. The body of the procedure is further indented one level to the right. The following is an example of an input text and its paragraphed output.

INPUT TEXT:

```
IFORMAT:PROCEDURE(NUMBER,WIDTH)CHARACTER(STRINGLIMIT);
DECLARE NUMBER FIXED;DECLARE WIDTH FIXED;
DECLARE L FIXED;DECLARE STRING CHARACTER(STRINGLIMIT);
STRING=NUMBER;L=LENGTH(STRING);IF L<WIDTH THEN
STRING=SUBSTR(X80,0,WIDTH-L)??STRING;RETURN STRING;
END IFORMAT;
```

PARAGRAPHED OUTPUT:

```
IFORMAT:
   PROCEDURE(NUMBER, WIDTH) CHARACTER(STRINGLIMIT);
      DECLARE NUMBER FIXED;
      DECLARE WIDTH FIXED;
      DECLARE L FIXED;
      DECLARE STRING CHARACTER(STRINGLIMIT);

      STRING = NUMBER;
      L = LENGTH(STRING);
      IF L < WIDTH THEN
         STRING = SUBSTR(X80, 0, WIDTH - L) ?? STRING;
      RETURN STRING;
   END IFORMAT;
```

3.4.2 PARAGRAPHING TO IMPROVE READABILITY

To acheive readability in the paragraphed output we impose certain restrictions without taking away all the freedom a programmer has to write programs in a readable form. This is achieved by different handling of declare statements, comments, macros, and blank lines, and to some extent operator precedence. They are discussed separately in the following sections.

3.4.2.1 DECLARE STATEMENTS

Each declare statement is printed on a new line. If more than one variable is declared in the same declare statement, then each variable will go on a new line and will be aligned. If a variable is initialized with a value in the declare statement, then this value is put on the same line as the rest of the declare statement. On the other hand, if an array variable is initialized with values, then these values are listed on the next line(s) and are indented one level to the right. The following is an example of an input text and its paragraphed output.

INPUT TEXT:

```
DECLARE VARIABLE0 FIXED;
DECLARE VARIABLE1 FIXED,VARIABLE2 CHARACTER(10),
VARIABLE3 FIXED;
DECLARE(VARIABLE4,VARIABLE5,VARIABLE6)FIXED,
VARIABLE7 FIXED;
DECLARE VARIABLE8 FIXED INITIAL (0), VARIABLE9(10)
FIXED INITIAL(0,1,2,3,4,5,6,7,8,9,10);
```

PARAGRAPHED OUTPUT:

```
DECALRE  VARIABLE0 FIXED;
DECLARE  VARIABLE1 FIXED,
         VARIABLE2 CHARACTER(10),
         VARIABLE3 FIXED;
DECLARE (VARIABLE4,
         VARIABLE5,
         VARIABLE6) FIXED,
         VARIABLE7 FIXED;
DECLARE  VARIABLE8 FIXED INITIAL(0),
         VARIABLE9(10) FIXED INITIAL(
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

3.4.2.2 COMMENTS

Comments are not part of the language but, when included in the program, they make it more readable. A well commented program is easier to understand. Different programmers have different styles of commenting their programs. Sometime it is preferable to put the heading of a program (or a module in the program) in the centre of the output page. More often, it is desirable to put a short comment at the end of the statement or at the end of the procedure name defining the function it performs. Whatever the case may be, we prefer that comments should not be paragraphed. We propose the following rules for handling comments.

- if the entire input line (card image) has nothing but a comment then the position of the comment is preserved and it is printed as it is.

- if the input line has a statement followed by a comment then the statement is paragraphed by the rules of logical structure and readability, as mentioned above, and the position of the comment with respect to the logical statement is preserved. This means that if there are, for example, five blanks between a logical statement and the comment, then the paragraphed listing will have the logical statement, followed by five blanks, followed by the comment.

- if the comment is embedded within the logical statement, then its position within the logical statement is preserved while the statement is paragraphed according to the paragraphing rules.

The following is an example of an input text and its paragraphed output.

INPUT TEXT:

```
                /******************/
                /* TEST COMMENTS */
                /******************/
   DECLARE I FIXED;DECLARE J FIXED;DECLARE K FIXED;
I=10;  /* INITIALIZE I */ J=I+1;
   /* COMPUTE K */ K = J+I*J;K=K+5;
IF I>K  /* CONDITION */   THEN  /* TRUE */
OUTPUT="I IS GREATER THAN K";ELSE  /* FALSE */
OUTPUT="I IS NOT GREATER THAN K";
```

PARAGRAPHED OUTPUT:

```
                /******************/
                /* TEST COMMENTS */
                /******************/

DECLARE I FIXED;
DECLARE J FIXED;
DECLARE K FIXED;
I = 10;  /* INITIALIZE I */
J = I + 1;
   /* COMPUTE K */
K = J + I*J;
K = K + 5;
IF I > K  /* CONDITION */   THEN  /* TRUE*/
   OUTPUT = "I IS GREATER THAN K";
ELSE  /* FALSE */
   OUTPUT = "I IS NOT GREATER THAN K";
```

3.4.2.3 MACROS

Macros symbolically represents a character string (text), the string being part of the program whose function is well understood. Macros may contain other macros within themselves. Instead of these strings in the program, their respective symbolic names are used. This makes the program more readable and reduces its complexity. When the scanner recognizes these symbolic names, they are replaced by their respective character strings and then these strings are scanned. To preserve their basic aim, the present paragraphing technique does not expand macros on the paragraphed output.

The macros may have logical stucture in their text, which requires paragraphing. Since macros are not expanded on the output listing, the first token received from the macro text plays a role in the paragraphing. The paragraphing is deferred until the entire macro text is scanned. In the following example two macros 'THENN' and 'ELSEE' are used and are declared to represent the character strings 'THEN DO;' and 'END;ELSE DO;' respectively. These strings are also used in the example by themselves to expose the different handling of the macros by the proposed paragraphing technique.

Consider the macro name 'THENN' and its corresponding text 'THEN DO;'. When this text is used by itself, then the symbol 'THEN' will be appended into the output buffer by the paragrapher. It will also cause the indentation level to be incremented one level to the right and the output buffer will be printed out. As a result, the next output line will be indented one level to the right. The symbol 'DO' will be appended into the output buffer and will also cause the indentaion level to be incremented one level to the right. The symbol ';' will be appended to the output buffer and the output buffer will be forced to be printed out. As a net result, after paragraphing the text 'THEN DO;', the indentation level is incremented two levels to the right.

On the other hand, when the macro 'THENN' is used, the paragrapher will take the actions noted above upon receiving the symbol 'THEN' of the macro text; except, instead of appending the symbol 'THEN', the macro name 'THENN' will be appended into the output buffer. The rest of the symbols of the macro text (that is, 'DO' and ';') will go unnoticed as far as the paragraphed output is concerned. Therefore, as a net result after paragraphing the macro text, 'THEN DO;', the indentation level is incremented one level to the right. The following is an example of an input text and its paragraphed output.

INPUT TEXT:

```
DECLARE I FIXED INITIAL(10);DECLARE J FIXED
INITIAL(20);DECLARE THENN LITERALLY'THEN DO;',
ELSEE LITERALLY'END;ELSE DO;';
IF I>10 THEN DO;I=I-1;J=J-1;END;
ELSE DO;I=I+1;J=J+1;END;
OUTPUT= I ?? ' ' ?? J;
IF I>10 THENN I=I-1;J=J-1;
ELSEE I=I+1;J=J+1;END;OUTPUT=I??' '??J;
```

PARAGRAPHED OUTPUT:

```
DECLARE I FIXED INITIAL(10);
DECLARE J FIXED INITIAL(20);
DECLARE THENN LITERALLY 'THEN DO;',
        ELSEE LITERALLY 'END;ELSE DO;';
IF I > 10 THEN
   DO;
      I = I - 1;
      J = J - 1;
   END;
ELSE
   DO;
      I = I + 1;
      J = J + 1;
   END;
OUTPUT = I ?? ' ' ?? J;
IF I > 10 THENN
   I = I - 1;
   J = J - 1;
ELSEE
   I = I + 1;
   J = J + 1;
END;
OUTPUT = I ?? ' ' ?? J;
```

3.4.2.4 BLANK LINES

All the blank lines in the input text are retained in the paragraphed output. There is only one occasion when, if necessary, blank lines are inserted in the output according to the following rule:

> If there are two consecutive procedures in the input program and there are N blank lines between these procedures, then in the paragraphed output N blank lines will be printed if N > 2, otherwise 2 blank lines will be printed.

The following is an example of an input text and its paragraphed output.

```
INPUT TEXT:

PROC1:PROCEDURE;GLOBAL_VAR=1;RETURN;END PROC1;
PROC2:PROCEDURE;GLOBAL_VAR=2;RETURN;END PROC2;



PROC3:PROCEDURE;GLOBAL_VAR=3;RETURN 3;END PROC3;


PRAGRAPHED OUTPUT:

PROC1:
   PROCEDURE;
      GLOBAL_VAR = 1;
      RETURN;
   END PROC1;

PROC2:
   PROCEDURE;
      GLOBAL_VAR = 2;
      RETURN;
   END PROC2;


PROC3:
   PROCEDURE;
      GLOBAL_VAR = 3;
      RETURN;
   END PROC3;
```

3.4.2.5 OPERATOR PRECEDENCE

The evaluation of an expression is performed from left to right following the rules of the heirarchy of operators present in the expression. This process of expression evaluation can also be highlighted on the output listing. Here we are dealing with arithmetic operators only, and the language we are using has only four of them. These are -, +, *, and /. The operators * and / have higher priority than - and +. The operators - and + have the same priority; so do the operators * and /. This operator precedence is shown on the paragraphed output by printing a blank on either side of - and + operators, and no blank at all on either side of * and / operators. The following is an example of an input text and its paragraphed output.

INPUT TEXT:

```
A = B + C * D / E - F ;
```

PARAGRAPHED OUTPUT:

```
A = B + C*D/E - F;
```

3.4.3 ALGORITHM FOR PARAGRAPHING

A simple approach is used for paragraphing the program input to the compiler. This paragraphing technique is table-driven and is not automated like the error repair technique, for the reasons mentioned in section 3.4. Although the paragraphing module included in the XPL skeleton compiler is sufficient for most XPL-like languages, a minimal effort will be required to modify the paragraphing module for handling the features of a given language which do not exist in the XPL language. This technique could easily be automated if the features of readability, as discussed in section 3.4, are not desired.

The table for paragraphing called 'paragraph' is a one dimensional array whose size is the same as the number of terminal symbols in the grammar. The nth element (n = 1,...,Vt) of the paragraphing table represents the nth terminal symbol as listed in the vocabulary produced by the language analyzer. The contents of each element of the paragraphing table indicate the case number of a DO CASE statement which will be executed for its corresponding terminal symbol. There is a great possibility that in a given language more than one terminal symbol will require to be treated in the same fashion. Therefore, one case element (same piece of code) can handle more than one terminal symbol if in the paragraphing table their respective elements contain the same case statement number. Consequently, the main paragraphing routine consists of a DO CASE statement which handles all the cases. Selected elements of this case statement are listed below; the tokens inside the delimiters indicate when each case will be executed:

```
DO CASE PARAGRAPH(CURRENT_SYMBOL);
    CASE(0):  /* NOT USED */
        ;
    CASE(1):  /* _!_ */
        ;
    CASE(2):  /*  ;  */
        ;
    CASE(3):  /* DECLARE */
        ;
    CASE(4):  /*  ,  */
        ;
    CASE(5):  /* <IDENTIFIER> */
        ;
      .
      .
      .
    CASE(20):  /* RETURN CALL RECURSIVE NOT */
        ;
      .
      .
      .
      .
END_CASE;
```

The code associated with each case statement is not shown in the DO CASE statement listed above. The code for each case consists mainly of procedure calls to paragraphing utility routines which will be discussed shortly. The paragraphing module constructed in this manner will be able to expose the logical structure of the program input to the compiler. This is straightforward work and, therefore, could be automated easily. But to improve the readability of the paragraphed output, as discussed in section 3.4.2, more code is associated with each case statement and is specific to XPL or XPL-like languages. It will, therefore, will not be discussed in this thesis.

The following are the utility routines of the paragraphing module. Their functions are discussed briefly:

ADD_IN_OUTBUF:

This routine is called with two parameters, PARM1 and PARM2, and they could be one of the following:

- PARM1:
  - NO_SPACE
  - SPACE_BEFORE
  - SPACE_AFTER
  - SPACE_BEFORE_AFTER

- PARM2:
  - RESERVED
  - ID_OR_FUNCTION
  - CHAR_STRINGS
  - NUMBERS

The function of this routine is to append the specified character string at the end of the output buffer according to PARM1, which is self explanatory. PARM2 specifies the type of the character string to be appended. The actual character string to be appended is received by calling the procedure 'get_temp' with parameter PARM2.

GET_TEMP:

This routine is called by the procedure 'add_in_outbuf' with PARM2 as parameter. If PARM2 is 'reserved', then it implies that the terminal symbol is a reserved word and will be retrieved from the vocabulary and returned to the calling procedure. If PARM2 is 'id_or_function', then the identifier or the function name will be retrieved from the string table, where it was stored by the scanner, and returned to the calling procedure. For 'char_strings', the character string will be fetched from the string table. When the scanner scans a number, it stores its value in a variable. So, for 'numbers' as PARM2, the number value will be fetched from that variable and returned as character string to the calling procedure.

PRINT_OUT:

This routine is called by the routine 'add_in_outbuf' to print out the buffer if the character string to be appended cannot fit in the current buffer. This routine is also called from the main DO CASE statement of the paragraphing module, when the output buffer is required to be printed out at any time, to expose the logical structure of the program. When the output buffer is printed out, the buffer is initialized with the proper indentation.

CHAPTER 4

USE OF THE XPL SKELETON COMPILER AND DISCUSSION OF RESULTS

The proposed compiler writing system is broken into four major steps. The steps, or phases, are identified as the semantic phase, the parse phase, the error repair phase, and the merge phase. The first phase generates synthesis procedures, the next two phases generate tables. In the fourth phase, the tables and the synthesis procedures are merged manually with the XPL skeleton compiler to produce the final compiler for the given language.

The language analyzer has been modified to handle the first three phases (see Appendix A). The XPL skeleton compiler is used in the final phase to produce the compiler.

The compiler writing system presented in this thesis is composed of the following two major parts:

1) The SLR(1) language analyzer which generates the following:
   - Parsing tables
   - Error repair tables
   - Synthesis procedures

2) The XPL skeleton compiler which consists of the following:

- Scanner routines
- Error repair routines
- Paragraphing routines
- Standard semantic routines
  - HDG building routines
  - HDG traversing routines
- Parsing routines
- Code emission routines for the CDC 6400 computer

The merging of the results from the first part into the second part is done manually. The whole process of producing a compiler is shown in Fig. 4.1, and the data flow and control flow of the generated compiler is shown in Fig. 4.2.

MERGE

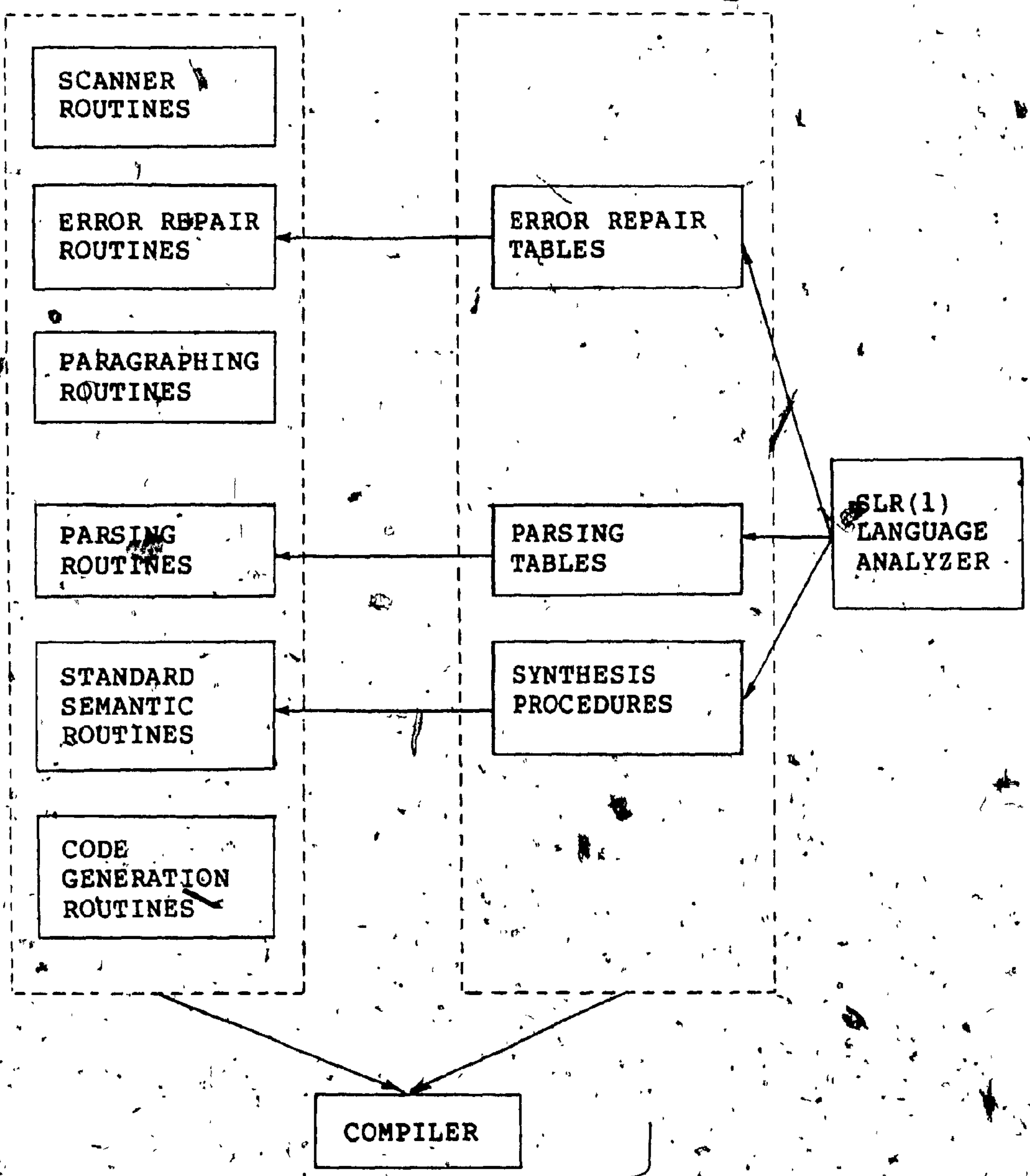
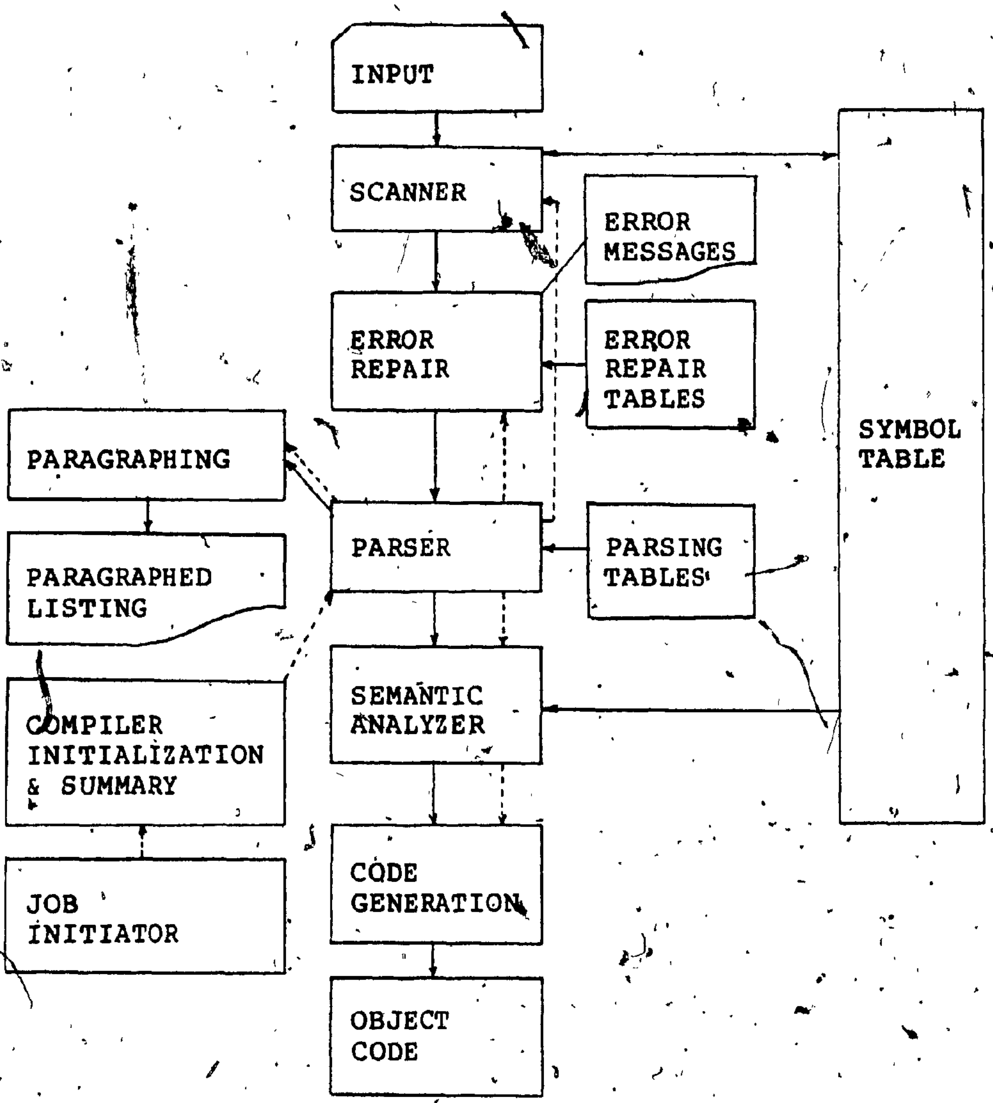

SCANNER ROUTINES
ERROR REPAIR ROUTINES
PARAGRAPHING ROUTINES
PARSING ROUTINES
STANDARD SEMANTIC ROUTINES
CODE GENERATION ROUTINES
ERROR REPAIR TABLES
PARSING TABLES
SYNTHESIS PROCEDURES
SLR(1) LANGUAGE ANALYZER
COMPILER

FIG. 4.1



LEGEND:

———> DATA FLOW
------> CONTROL FLOW

FIG. 4.2

The major modules of the XPL skeleton compiler were extracted from the existing XPL compiler at Concordia University. These modules include the scanning routines, standard semantic routines, parsing routines, and the code emission routines. The error repair routines and the paragraphing routines were coded and included in the skeleton compiler to support the work presented in this thesis. Minor changes were made in the scanning routines to make them compatible with the paragraphing routines. One of the parsing routines in the parser was also slightly modified to allow the parser to give control to the error repair and paragraphing routines. The nucleus of this modified routine is listed in section 3.3.4. The transfer of the control from parser to error repair and paragraphing routines is discussed in section 3.3.4 and section 3.4.3 respectively.

The present state of the XPL skeleton compiler is quite sufficient to produce compilers for different pedagogical programming languages. It may be necessary to augment the existing semantic routines in the skeleton compiler to implement certain special features of the language. This will require manual work.

Depending on the grammar and compiler implementation restrictions, the following routines may require modification:

- Scanning routines
- Error repair routines
- Paragraphing routines

The scanning routines will require modification if, for example, it is desired to write comments inside the program by any method other than enclosing comments between '/*' and '*/'. Similarly, if it is desired not to allow string constants in a program to continue on the next line (card), the scanning routines must be modified.

The part of the error repair routines which handles the grouping elements may be modified or re-written, if so required. It is the user's (that is, the compiler writer or the language designer) responsibility to write the procedure which handles the grouping elements. This is discussed in detail in appendix A.

The paragraphing technique presented in this thesis is explicit to the XPL language. As discussed earlier in section 3.4, this technique is not automated due to the special cases regarding readability of the programs on the output listings.

For a given language, if the set of its terminal symbols (VLt) is a proper subset of the terminal symbols of the XPL language (VXt), then only the paragraphing table needs modification. It may be noted that the amount of work involved is negligible. On the other hand, if VLt is not a proper subset of VXt, then the code for the new terminal symbols must be written and included in the paragraphing module.

We believe that the terminal symbols and their usage in the XPL grammar is quite similar to those used in other commercial languages. Therefore, it is safe to state that the paragraphing module included in the XPL skeleton compiler is sufficient, and if any modification is required it will be minimal.

In the rest of this chapter we will present the results from the first part of the proposed compiler writing system (i.e., the SLR(1) language analyzer). We have used the XPL language itself and implemented most of its features in the generated compiler to support all the proposed features of the error repair and paragraphing techniques. The ease of producing the compiler, with better error repair and paragraphing by the proposed compiler writing system, will be evident by the results produced by the SLR(1) language analyzer. These include the parsing tables, error repair tables, and the synthesis procedures.

The input to and the output from the SLR(1) language analyzer is presented in the following pages. The procedure given in appendix A is used to supply the input and to obtain the output from the analyzer.

1) INPUT DATA TO THE SLR(1) LANGUAGE ANALYZER FOR THE GROUPING_TABLE AND THE ACTION_TABLE.

(Detailed description is given in Appendix A.)

```
(           1
)           2
PROCEDURE   3
DO          4
END         5
;           6
FIXED       6
CHARACTER   6
THEN        6
END_PROC    7
||          8
EOD
;               PROCEDURE      REPLACE_CURRENT_SYMBOL    $
;               RECURSIVE      REPLACE_CURRENT_SYMBOL    $
;               EXIT           INSERT                    CALL
,               DECLARE        REPLACE_CURRENT_SYMBOL    ;
,               PROCEDURE      REPLACE_CURRENT_SYMBOL    $
,               RECURSIVE      REPLACE_CURRENT_SYMBOL    $
,               END_PROC       REPLACE_CURRENT_SYMBOL    ;
,               EXIT           REPLACE_CURRENT_SYMBOL    ;
,               IF             REPLACE_CURRENT_SYMBOL    ;
,               ELSE           REPLACE_CURRENT_SYMBOL    ;
,               OUTPUT         REPLACE_CURRENT_SYMBOL    ;
,               DO             REPLACE_CURRENT_SYMBOL    ;
,               END            REPLACE_CURRENT_SYMBOL    ;
,               RETURN         REPLACE_CURRENT_SYMBOL    ;
,               CALL           REPLACE_CURRENT_SYMBOL    ;
<IDENTIFIER>    |              INSERT                    ;
<IDENTIFIER>    <IDENTIFIER>   INSERT                    ;
<IDENTIFIER>    PROCEDURE      INSERT                    $
<IDENTIFIER>    RECURSIVE      INSERT                    $
<IDENTIFIER>    END_PROC       INSERT                    ;
<IDENTIFIER>    EXIT           INSERT                    ;
<IDENTIFIER>    IF             INSERT                    ;
<IDENTIFIER>    ELSE           INSERT                    ;
<IDENTIFIER>    OUTPUT         INSERT                    ;
<IDENTIFIER>    FILE           INSERT                    ;
<IDENTIFIER>    DO             INSERT                    ;
<IDENTIFIER>    END            INSERT                    ;
<IDENTIFIER>    RETURN         INSERT                    ;
<IDENTIFIER>    CALL           INSERT                    ;
<STRING>        |              INSERT                    ;
<STRING>        DECLARE        INSERT                    ;
<STRING>        <IDENTIFIER>   INSERT                    ;
<STRING>        ENTRY          INSERT                    ;
<STRING>        RETURNS        INSERT                    ;
<STRING>        $              REPLACE_NEXT_SYMBOL       ;
<STRING>        END_PROC       INSERT                    ;
<STRING>        EXIT           INSERT                    ;
<STRING>        IF             INSERT                    ;
<STRING>        ELSE           INSERT                    ;
<STRING>        OUTPUT         INSERT                    ;
<STRING>        FILE           INSERT                    ;
<STRING>        DO             INSERT                    ;
```

```
<STRING>      END             INSERT                   ;
<STRING>      RETURN          INSERT                   ;
<STRING>      CALL            INSERT                   ;
)             !               INSERT                   ;
)             <IDENTIFIER>    INSERT                   ;
)             ENTRY           INSERT                   ;
)             RETURNS         INSERT                   ;
)             $               REPLACE_NEXT_SYMBOL      ;
)             PROCEDURE       REPLACE_CURRENT_SYMBOL   $
)             RECURSIVE       REPLACE_CURRENT_SYMBOL   $
)             END_PROC        INSERT                   ;
)             EXIT            INSERT                   ;
)             IF              INSERT                   ;
)             ELSE            INSERT                   ;
)             OUTPUT          INSERT                   ;
)             DO              INSERT                   ;
)             END             INSERT                   ;
)             RETURN          INSERT                   ;
)             CALL            INSERT                   ;
FIXED         DECLARE         INSERT                   ;
FIXED         <IDENTIFIER>    INSERT                   ;
FIXED         $               REPLACE_NEXT_SYMBOL      ;
CHARACTER     <NUMBER>        INSERT                   (
INITIAL       <NUMBER>        INSERT                   (
$             DECLARE         REPLACE_CURRENT_SYMBOL   ;
$             <IDENTIFIER>    REPLACE_CURRENT_SYMBOL   ;
$             ENTRY           REPLACE_CURRENT_SYMBOL   ;
$             END_PROC        REPLACE_CURRENT_SYMBOL   ;
$             EXIT            REPLACE_CURRENT_SYMBOL   ;
$             IF              REPLACE_CURRENT_SYMBOL   ;
$             ELSE            REPLACE_CURRENT_SYMBOL   ;
$             OUTPUT          REPLACE_CURRENT_SYMBOL   ;
$             FILE            REPLACE_CURRENT_SYMBOL   ;
$             DO              REPLACE_CURRENT_SYMBOL   ;
$             END             REPLACE_CURRENT_SYMBOL   ;
$             RETURN          REPLACE_CURRENT_SYMBOL   ;
$             CALL            REPLACE_CURRENT_SYMBOL   ;
PROCEDURE     DECLARE         INSERT                   ;
PROCEDURE     ,               REPLACE_NEXT_SYMBOL      ;
PROCEDURE     <IDENTIFIER>    INSERT                   ;
PROCEDURE     END_PROC        INSERT                   ;
PROCEDURE     EXIT            INSERT                   ;
PROCEDURE     IF              INSERT                   ;
PROCEDURE     OUTPUT          INSERT                   ;
PROCEDURE     FILE            INSERT                   ;
PROCEDURE     DO              INSERT                   ;
PROCEDURE     END             INSERT                   ;
PROCEDURE     RETURN          INSERT                   ;
PROCEDURE     CALL            INSERT                   ;
RECURSIVE     ;               INSERT                   PROCEDURE
RECURSIVE     $               INSERT                   PROCEDURE
END_PROC      !               INSERT                   ;
END_PROC      $               REPLACE_NEXT_SYMBOL      ;
END_PROC      END_PROC        INSERT                   ;
END_PROC      EXIT            INSERT                   ;
END_PROC      IF              INSERT                   ;
END_PROC      ELSE            INSERT                   ;
END_PROC      OUTPUT          INSERT                   ;
END_PROC      FILE            INSERT                   ;
END_PROC      DO              INSERT                   ;
END_PROC      END             INSERT                   ;
END_PROC      RETURN          INSERT                   ;
END_PROC      CALL            INSERT                   ;
OUTPUT        <IDENTIFIER>    INSERT                   (
OUTPUT        <NUMBER>        INSERT                   (
DO            $               REPLACE_NEXT_SYMBOL      ;
DO            IF              INSERT                   ;
DO            RETURN          INSERT                   ;
DO            CALL            INSERT                   ;
END           _!              INSERT                   ;
END           <IDENTIFIER>    INSERT                   ;
END           $               REPLACE_NEXT_SYMBOL      ;
END           END_PROC        INSERT                   ;
END           EXIT            INSERT                   ;
END           IF              INSERT                   ;
END           ELSE            INSERT                   ;
END           OUTPUT          INSERT                   ;
END           FILE            INSERT                   ;
END           DO              INSERT                   ;
END           END             INSERT                   ;
END           RETURN          INSERT                   ;
RETURN        $               REPLACE_NEXT_SYMBOL      ;
RETURN        END_PROC        INSERT                   ;
RETURN        IF              INSERT                   ;
RETURN        ELSE            INSERT                   ;
RETURN        OUTPUT          INSERT                   ;
RETURN        FILE            INSERT                   ;
RETURN        DO              INSERT                   ;
RETURN        END             INSERT                   ;
RETURN        CALL            INSERT                   )
RETURN        GENERATE        INSERT                   )
RETURN        INLINE          INSERT                   ;
<NUMBER>      _!              INSERT                   ;
<NUMBER>      <IDENTIFIER>    INSERT                   ;
<NUMBER>      (               INSERT                   +
<NUMBER>      FIXED           INSERT                   )
<NUMBER>      CHARACTER       INSERT                   )
<NUMBER>      INITIAL         INSERT                   )
<NUMBER>      END_PROC        INSERT                   ;
<NUMBER>      EXIT            INSERT                   ;
<NUMBER>      IF              INSERT                   ;
<NUMBER>      ELSE            INSERT                   ;
<NUMBER>      OUTPUT          INSERT                   ;
<NUMBER>      FILE            INSERT                   ;
<NUMBER>      DO              INSERT                   ;
<NUMBER>      END             INSERT                   ;
<NUMBER>      RETURN          INSERT                   ;
<NUMBER>      CALL            INSERT                   ;
<NUMBER>      <NUMBER>        INSERT                   +
```

2) OUTPUT FROM THE SLR(1) LANGUAGE ANALYZER.

The input augmented grammar is also listed by the analyzer.

(Detailed description is given in Appendix A.)

SLR(1) GRAMMAR ANALYSER   - SGWU VERSION 2.0

```
T
F
E
D
P
M
S
A
F    SYNO
D          DECLARE(PROD,I,OFFSET,INITIAL_PP,INITIAL_DP1,DP1_LIMIT)FIXED;
R <PROGRAM> <MAIN BODY>
S               CALL MAIN_PROG;
R <MAIN BODY> <DECLARATION_LIST> <STATEMENT LIST>
S               G1(MP) = G1(SP);
R   <DECLARATION LIST>
S               G1(MP) = NODE1(NOP);
R   <STATEMENT LIST>
R <DECLARATION LIST> <DECLARATION>
R   <DECLARATION LIST> <DECLARATION>
R <DECLARATION> <DECLARATION STATEMENT> ;
R   <PROCEDURE DEFINITION> ;
R <DECLARATION STATEMENT> DECLARE <DECLARATION ELEMENT LIST>
R <DECLARATION ELEMENT LIST> <DECLARATION ELEMENT>
R   <DECLARATION ELEMENT LIST> , <DECLARATION ELEMENT>
R <DECLARATION ELEMENT> <IDENTIFIER> LITERALLY <STRING>
S               CALL DEFINE_MACRO(VAR(MP),VAR(SP));
R   <IDENTIFIER SPECIFICATION>
R   <IDENTIFIER SPECIFICATION> <INITIAL LIST>
S               IF INITIAL_LIMIT < PP  THENN
S                  CALL WARNING('TOO MANY INITIAL VALUES');
S               END;
S               PP = INITIAL_LIMIT;
S               CALL PURGE_STRING_TABLE;
R <IDENTIFIER SPECIFICATION> <IDENTIFIER> <TYPE>
S               CALL ENTER_ID(SCALAR_DCL,0);
R   <IDENTIFIER> <DIMENSION> <TYPE>
S               CALL ENTER_ID(ARRAY_DCL,VAL(MPP1));
R   ( <IDENTIFIER LIST> ) <TYPE>
S               CALL ENTER_ID_LIST(SCALAR_DCL,0);
R   ( <IDENTIFIER LIST> ) <DIMENSION> <TYPE>
S               CALL ENTER_ID_LIST(ARRAY_DCL,VAL(SP-1));
R   ( <BIT FIELD LIST> ) <DIMENSION> <TYPE>
S               INITIAL_TYPE = TYPE(SP);
S               IF TYPE(SP) = FIXEDTYPE THENN
S                  CALL ALIGN;
S                  INITIAL_START = PP; /* LOWER LIMIT */
S                  INITIAL_PP = PP;
S                  INITIAL_DP1 = DP1;
S                  OFFSET = 0;
S                  PARALLEL_ALLOCATED = FALSE;
```

```
R <LOGICAL SECONDARY> <LOGICAL PRIMARY>
R   NOT <LOGICAL PRIMARY>
S                   CALL EVALUATE_UNARY(42); /* NOT */
R <LOGICAL PRIMARY> <STRING EXPRESSION>
R   <STRING EXPRESSION> <RELATION> <STRING EXPRESSION>
S                   CALL EVALUATE(VAL(MPP1)); /* <RELATION */
R <RELATION> =
S                   VAL(MP) = 20;
R   <
S                   VAL(MP) = 21;
R   >
S                   VAL(MP) = 22;
R   < =
S                   VAL(MP) = 23;
R   >=
S                   VAL(MP) = 24;
R   NE
S                   VAL(MP) = 25;
R <STRING EXPRESSION> <ARITHMETIC EXPRESSION>
R   <STRING EXPRESSION> | | <ARITHMETIC EXPRESSION>
S                   CALL EVALUATE(50); /* CONCATENATE */
R <ARITHMETIC EXPRESSION> <TERM>
R   <ARITHMETIC EXPRESSION> <ADD> <TERM>
S                   CALL EVALUATE(VAL(MPP1));
R   <ADD> <TERM>
S                   CALL EVALUATE_UNARY(VAL(MP));
R <ADD> +
S                   VAL(MP) = ADD;
R   -
S                   VAL(MP) = SUBT;
R <TERM> <PRIMARY>
R   <TERM> <MULT> <PRIMARY>
S                   CALL EVALUATE(VAL(MPP1));
R <MULT> *
S                   VAL(MP) = MULT;
R   /
S                   VAL(MP) = DIV;
R   MOD
S                   VAL(MP) = MODD ;
R <PRIMARY> <CONSTANT>
S                   CLASS(MP) = CONSTANT_CLASS;
R   <VARIABLE>
R   ( <EXPRESSION> )
S                   CALL MOVE_STACKS(MPP1,MP);
R <VARIABLE> <IDENTIFIER>
S                   CALL ACCESS_VARIABLE(0);
R   <IDENTIFIER> ( <EXPRESSION LIST> )
S                   EXP_CNT = FIXL(SP-1);
S                   CALL ACCESS_VARIABLE(CNT(SP-1));
R   <BUILTIN FUNCTION>
S                   CALL EVALUATE_BUILTIN_FUNCTION(0);
R   <BUILTIN FUNCTION> ( <EXPRESSION LIST> )
S                   EXP_CNT = FIXL(SP-1);
S                   CALL EVALUATE_BUILTIN_FUNCTION(CNT(SP-1));
```

```
R <CASE BODY> <STATEMENT>
S                 CNT(MP) = 1; /* FIRST CASE */
S                 INFORMATION = ' CASE 1';
S                 G1(MP) = SHORT_STATEMENT(NOP,G1(MP),CARDNUM(MP));
R   <CASE BODY> <STATEMENT>
S                 CNT(MP) = CNT(MP) + 1;
S                 INFORMATION = ' CASE '||CNT(MP);
S                 G1(SP) = SHORT_STATEMENT(NOP,G1(SP),CARDNUM(MP));
S                 G1(MP) = LINK_CHAIN(G1(MP),G1(SP));
R <RETURN STATEMENT> RETURN
S                 G1(MP) = SPECIAL(RTRN,PROC_INDEX,CARDNUM(MP));
R   RETURN <EXPRESSION>
S                 IF RETURNED_TYPE = CHRTYPE AND TYPE(SP) = FIXEDTYPE THENN
S                    CALL EXP_CONVERT(SP);
S                 END;
S                 IF RETURNED_TYPE NE TYPE(SP) THENN
S                    CALL ERRO[illegible]EGAL RETURN TYPE');
S                 ELSEE
S                    G1(MP) [illegible]NG_STATEMENT(RTRNX,-1,FORCE_GRAPH(SP),
S                                         PROC_INDEX,0,CARDNUM(MP));
S                 END;
R <CALL STATEMENT> CALL <IDENTIFIER>
S                 G1(MP) = GENERATE_CALL(ID_LOOKUP(MPP1),-1,CALLP);
S                 G1(MP) = SHORT_STATEMENT(NOP,G1(MP),CARDNUM(MP));
R   CALL <IDENTIFIER> ( <EXPRESSION LIST> )
S                 EXP_CNT = FIXL(SP-1);
S                 I = ID_LOOKUP(MPP1);
S                 IF I NE APPEND_STRING THENN
S                    G1(MP) = GENERATE_CALL(I,CNT(SP-1),CALLP);
S                    G1(MP) = SHORT_STATEMENT(NOP,G1(MP),CARDNUM(MP));
S                 ELSEE
S                    CALL GENERATE_APPEND(CNT(SP-1));
S                 END;
R   CALL EXIT
S                 G1(MP) = SPECIAL(SRTRN,0,CARDNUM(MP));
R <GENERATE BLOCK> <GENERATE HEAD> <INLINE BODY> END
R   <GENERATE HEAD> <INLINE BODY> <LABEL> END
R <GENERATE HEAD> GENERATE ;
R <INLINE BODY> <INLINE STATEMENT>
R   <INLINE BODY> <INLINE STATEMENT>
R <INLINE STATEMENT> INLINE ( <EXPRESSION LIST> ) ;
R   <LABEL>  INLINE ( <EXPRESSION LIST> ) ;
E
P    SYN2
D           DECLARE PROD FIXED;
R <LABEL> <IDENTIFIER> :
R <EXPRESSION> <LOGICAL FACTOR>
R   <EXPRESSION> OR <LOGICAL FACTOR>
S                 CALL EVALUATE(41); /* OR */
R <LOGICAL FACTOR> <LOGICAL SECONDARY>
R   <LOGICAL FACTOR> AND <LOGICAL SECONDARY>
S                 CALL EVALUATE(40); /* AND */
```

```
S                   CALL EXPAND_EXP(EXP_CNT+2,MPP1);
S                   IF TYPE(MP) = FIXEDTYPE AND TYPE(MPP1) = FIXEDTYPE THENN
S                      G1(MP) = EXPRESSION_LIST(FORCE_GRAPH(MP),
S                                                FORCE_GRAPH(MPP1),-1);
S                      G1(MP) = LONG_STATEMENT(75,G1(MP),NODE2(LA,K),0,0,
S                                                            CARDNUM(MP));
S                   ELSEE
S                      CALL ERROR('FILE PARAMETERS MUST BE FIXED TYPE');
S                   END;
S                ELSEE
S                   CALL ERROR('FILE NEEDS TWO PARAMETERS');
S                END;
R   FILE ( <EXPRESSION LIST> ) = <IDENTIFIER> ( <EXPRESSION> )
R   <IDENTIFIER> = FILE ( <EXPRESSION LIST> )
S                EXP_CNT = FIXL(SP-1);
S                IF CNT(SP-1) = 2 THENN /* 2 EXPRESSIONS */
S                   K = ID_LOOKUP(MP);
S                   CALL EXPAND_EXP(EXP_CNT+1,MPP1);
S                   CALL EXPAND_EXP(EXP_CNT+2,SP);
S                   IF TYPE(MPP1) = FIXEDTYPE AND TYPE(SP) = FIXEDTYPE THENN
S                      G1(MP) = EXPRESSION_LIST(FORCE_GRAPH(MPP1),FORCE_GRAPH(
S                         SP),-1);
S                      G1(MP) = LONG_STATEMENT(74,G1(MP),NODE2(LA,K),0,0,
S                                                            CARDNUM(MP));
S                   ELSEE
S                      CALL ERROR('FILE PARAMETERS MUST BE FIXED TYPE');
S                   END;
S                ELSEE
S                   CALL ERROR('FILE NEEDS TWO PARAMETERS');
S                END;
R   <IDENTIFIER> ( <EXPRESSION> ) = FILE ( <EXPRESSION LIST> )
R <GROUP> DO ; <STATEMENT LIST> END
S                G1(MP) = G1(MP+2);
R   DO WHILE <EXPRESSION> ; <STATEMENT LIST> END
S                G1(MP) = DOWHILE(FORCE_GRAPH(MP+2),G1(SP-1),NOP,CARDNUM(MP));
R   DO <IDENTIFIER> = <EXPRESSION> TO <EXPRESSION> ; <STATEMENT LIST> END
S                I = ID_LOOKUP(MPP1);
S                G1(MP) = LONG_STATEMENT(ST,-1,FORCE_GRAPH(MP+3),I,0,
S                                        CARDNUM(MP) );
S                J = EMIT_WORD(0);
S                G1(MPP1) = LONG_STATEMENT(STW,-1,FORCE_GRAPH(MP+5),
S                                        0,SHR(J,2),CARDNUM(MP));
S                G1(MP) = LINK_CHAIN(G1(MP),G1(MPP1));
S                G1(MPP1)=BINARY_EXPRESSION(23,NODE2(L,I),NODE3(LW,0,SHR(J,2))
S                      ,0,0);
S                G1(SP) = BINARY_EXPRESSION(ADD,NODE2(L,I),NODE2(LIT,1),0,0);
S                G1(SP) = LONG_STATEMENT(ST,-1,G1(SP),I,0,CARDNUM(MP));
S                G1(MPP1) = DOWHILE(G1(MPP1),G1(SP-1),G1(SP),CARDNUM(MP));
S                G1(MP) = LINK_CHAIN(G1(MP),G1(MPP1));
R   DO CASE <EXPRESSION> ; <CASE BODY> END
S                G1(MP) = DOCASE(FORCE_GRAPH(MP+2),G1(SP-1),NOP,CARDNUM(MP));
S                INFORMATION = '';
```

```
S                 CALL EXPAND_EXP(EXP_CNT+2,MPP1);
S                 IF TYPE(MP) = FIXEDTYPE AND TYPE(MPP1) = FIXEDTYPE THENN
S                    G1(MP) = EXPRESSION_LIST(FORCE_GRAPH(MP),
S                                         FORCE_GRAPH(MPP1),-1);
S                    G1(MP) = LONG_STATEMENT(75,G1(MP),NODE2(LA,K),0,0,
S                                                     CARDNUM(MP));
S                 ELSEE
S                    CALL ERROR('FILE PARAMETERS MUST BE FIXED TYPE');
S                 END;
S              ELSEE
S                 CALL ERROR('FILE NEEDS TWO PARAMETERS');
S              END;
R  FILE ( <EXPRESSION LIST> ) = <IDENTIFIER> ( <EXPRESSION> )
R  <IDENTIFIER> = FILE ( <EXPRESSION LIST> )
S              EXP_CNT = FIXL(SP-1);
S              IF CNT(SP-1) = 2 THENN /* 2 EXPRESSIONS */
S                 K = ID_LOOKUP(MP);
S                 CALL EXPAND_EXP(EXP_CNT+1,MPP1);
S                 CALL EXPAND_EXP(EXP_CNT+2,SP);
S                 IF TYPE(MPP1) = FIXEDTYPE AND TYPE(SP) = FIXEDTYPE THENN
S                    G1(MP) = EXPRESSION_LIST(FORCE_GRAPH(MPP1),FORCE_GRAPH(
S                       SP),-1);
S                    G1(MP) = LONG_STATEMENT(74,G1(MP),NODE2(LA,K),0,0,
S                                                     CARDNUM(MP));
S                 ELSEE
S                    CALL ERROR('FILE PARAMETERS MUST BE FIXED TYPE');
S                 END;
S              ELSEE
S                 CALL ERROR('FILE NEEDS TWO PARAMETERS');
S              END;
R  <IDENTIFIER> ( <EXPRESSION> ) = FILE ( <EXPRESSION LIST> )
R <GROUP> DO ; <STATEMENT LIST> END
S              G1(MP) = G1(MP+2);
R  DO WHILE <EXPRESSION> ; <STATEMENT LIST> END
S              G1(MP) = DOWHILE(FORCE_GRAPH(MP+2),G1(SP-1),NOP,CARDNUM(MP));
R  DO <IDENTIFIER> = <EXPRESSION> TO <EXPRESSION> ; <STATEMENT LIST> END
S              I = ID_LOOKUP(MPP1);
S              G1(MP) = LONG_STATEMENT(ST,-1,FORCE_GRAPH(MP+3),I,0,
S                                     CARDNUM(MP) );
S              J = EMIT_WORD(0);
S              G1(MPP1) = LONG_STATEMENT(STW,-1,FORCE_GRAPH(MP+5),
S                                     0,SHR(J,2),CARDNUM(MP));
S              G1(MP) = LINK_CHAIN(G1(MP),G1(MPP1));
S              G1(MPP1)=BINARY_EXPRESSION(23,NODE2(L,I),NODE3(LW,0,SHR(J,2))
S                  ,0,0);
S              G1(SP) = BINARY_EXPRESSION(ADD,NODE2(L,I),NODE2(LIT,1),0,0);
S              G1(SP) = LONG_STATEMENT(ST,-1,G1(SP),I,0,CARDNUM(MP));
S              G1(MPP1) = DOWHILE(G1(MPP1),G1(SP-1),G1(SP),CARDNUM(MP));
S              G1(MP) = LINK_CHAIN(G1(MP),G1(MPP1));
R  DO_CASE <EXPRESSION> ; <CASE BODY> END
S              G1(MP) = DOCASE(FORCE_GRAPH(MP+2),G1(SP-1),NOP,CARDNUM(MP));
S              INFORMATION = '.';
```

```
R <STATEMENT LIST> <STATEMENT>
R   <STATEMENT LIST> <STATEMENT>
S                  G1(MP) = LINK_CHAIN(G1(MP),G1(SP));
R <STATEMENT> <BASIC STATEMENT>
R   <IF STATEMENT>
R <BASIC STATEMENT> <ASSIGNMENT> ;
R   <GROUP> ;
R   <RETURN STATEMENT> ;
R   <CALL STATEMENT> ;
R   EXIT ;
R   <GENERATE BLOCK> ;
R   ;
S                  G1(MP) = SHORT_STATEMENT(NOP,-1,CARDNUM(MP));
R <IF STATEMENT> <IF CLAUSE> <STATEMENT>
S                  IF CLASS(MP) = CONSTANT_CLASS THENN
S                     IF VAL(MP) = 1 THENN
S                        G1(MP) = G1(SP);
S                     END;
S                  ELSEE
S                     G1(MP) = IFTHENELSE(G1(MP),NOP,G1(SP),CARDNUM(MP));
S                  END;
R   <IF CLAUSE> <TRUE PART> <STATEMENT>
S                  IF CLASS(MP) = CONSTANT_CLASS THENN
S                     IF VAL(MP) = 1 THENN
S                        G1(MP) = G1(MPP1);
S                     ELSEE
S                        G1(MP) = G1(SP);
S                     END;
S                  ELSEE
S                     G1(MP) = IFTHENELSE(G1(MP),G1(SP),G1(MPP1),CARDNUM(MP));
S                  END;
R <IF CLAUSE> IF <EXPRESSION> THEN
S                  CALL MOVE_STACKS(MPP1,MP);
R <TRUE PART> <BASIC STATEMENT> ELSE
R <ASSIGNMENT> <IDENTIFIER> = <EXPRESSION>
S                  CALL GEN_STORE(FALSE);
R   <IDENTIFIER> ( <EXPRESSION> ) = <EXPRESSION>
S                  CALL GEN_STORE(TRUE);
R   OUTPUT = <EXPRESSION>
S                  IF TYPE(SP) = FIXEDTYPE THENN
S                     CALL EXP_CONVERT(SP);
S                  END;
S                  G1(MP) = LONG_STATEMENT(71,-1,FORCE_GRAPH(SP),0,0,CARDNUM(MP));
R   OUTPUT ( <EXPRESSION> ) = <EXPRESSION>
S                  IF TYPE(SP) = FIXEDTYPE THENN
S                     CALL EXP_CONVERT(SP);
S                  END;
S                  G1(MP) = LONG_STATEMENT(71,FORCE_GRAPH(MP+2),
S                                          FORCE_GRAPH(SP),1,0,CARDNUM(MP));
R   FILE ( <EXPRESSION LIST> ) = <IDENTIFIER>
S                  EXP_CNT = FIXL(MP+2);
S                  IF CNT(MP+2) = 2 THENN /* 2 EXPRESSIONS */
S                     K = ID_LOOKUP(SP);
S                     CALL EXPAND_EXP(EXP_CNT+1,MP);
```

```
R <STATEMENT LIST> <STATEMENT>
R   <STATEMENT LIST> <STATEMENT>
S                 G1(MP) = LINK_CHAIN(G1(MP),G1(SP));
R <STATEMENT> <BASIC STATEMENT>
R   <IF STATEMENT>
R <BASIC STATEMENT> <ASSIGNMENT> ;
R   <GROUP> ;
R   <RETURN STATEMENT> ;
R   <CALL STATEMENT> ;
R   EXIT ;
R   <GENERATE BLOCK> ;
R   ;
S                 G1(MP) = SHORT_STATEMENT(NOP,-1,CARDNUM(MP));
R <IF STATEMENT> <IF CLAUSE> <STATEMENT>
S                 IF CLASS(MP) = CONSTANT_CLASS THENN
S                    IF VAL(MP) = 1 THENN
S                       G1(MP) = G1(SP);
S                    END;
S                 ELSEE
S                    G1(MP) = IFTHENELSE(G1(MP),NOP,G1(SP),CARDNUM(MP));
S                 END;
R   <IF CLAUSE> <TRUE PART> <STATEMENT>
S                 IF CLASS(MP) = CONSTANT_CLASS THENN
S                    IF VAL(MP) = 1 THENN
S                       G1(MP) = G1(MPP1);
S                    ELSEE
S                       G1(MP) = G1(SP);
S                    END;
S                 ELSEE
S                    G1(MP) = IFTHENELSE(G1(MP),G1(SP),G1(MPP1),CARDNUM(MP));
S                 END;
R <IF CLAUSE> IF <EXPRESSION> THEN
S                 CALL MOVE_STACKS(MPP1,MP);
R <TRUE PART> <BASIC STATEMENT> ELSE
R <ASSIGNMENT> <IDENTIFIER> = <EXPRESSION>
S                 CALL GEN_STORE(FALSE);
R   <IDENTIFIER> ( <EXPRESSION> ) = <EXPRESSION>
S                 CALL GEN_STORE(TRUE);
R   OUTPUT = <EXPRESSION>
S                 IF TYPE(SP) = FIXEDTYPE THENN
S                    CALL EXP_CONVERT(SP);
S                 END;
S                 G1(MP) = LONG_STATEMENT(71,-1,FORCE_GRAPH(SP),0,0,CARDNUM(MP));
R   OUTPUT ( <EXPRESSION> ) = <EXPRESSION>
S                 IF TYPE(SP) = FIXEDTYPE THENN
S                    CALL EXP_CONVERT(SP);
S                 END;
S                 G1(MP) = LONG_STATEMENT(71,FORCE_GRAPH(MP+2),
S                                         FORCE_GRAPH(SP),1,0,CARDNUM(MP));
R   FILE ( <EXPRESSION LIST> ) = <IDENTIFIER>
S                 EXP_CNT = FIXL(MP+2);
S                 IF CNT(MP+2) = 2 THENN /* 2 EXPRESSIONS */
S                    K = ID_LOOKUP(SP);
S                    CALL EXPAND_EXP(EXP_CNT+1,MP);
```

```
R <BIT SPECIFICATION> <IDENTIFIER> ( <CONSTANT VALUE> )
S                 CALL CHECK_CONSTANT;
S                 IF VAL(MP) > 60 THENN
S                    CALL ERROR('PRECISION TOO BIG');
S                    VAL(SP) = 60;
S                 END;
S                 VAL(MP) = VAL(SP);
R <TYPE LIST> <TYPE>
S                 ID_LIST_PTR = 0;
S                 ID_TYPE(0) = TYPE(MP);
S                 ID_WIDTH(0) = VAL(MP);
R   <TYPE LIST> , <TYPE>
S                 INC_ID_PTR;
S                 ID_TYPE(ID_LIST_PTR)= TYPE(SP);
S                 ID_WIDTH(ID_LIST_PTR) = VAL(SP);
R <INITIAL LIST> INITIAL ( <CONSTANT LIST> )
R <CONSTANT LIST> <CONSTANT VALUE>
S                 IF LEVELNO > 0 THENN
S                    CALL ERROR('INITIAL VALUES ILLEGAL IN RECURSIVE PROCEDURE');
S                 END;
S                 PP = INITIAL_START;
S                 CALL SETINIT(MP);
R   <CONSTANT LIST> , <CONSTANT VALUE>
S                 CALL SETINIT(SP);
R <CONSTANT VALUE> <EXPRESSION>
S                 IF CLASS(MP) NE CONSTANT_CLASS THENN
S                    CALL ERROR('ONLY CONSTANT EXPRESSIONS ALLOWED');
S                    VAL(MP) = 0;
S                    VAR(MP) = EMPTY_STRING_PTR;
S                 END;
R <PROCEDURE DEFINITION> <PROCEDURE HEAD> <MAIN BODY> <ENDING>
S                 CALL CLOSE_SCOPE;
R <PROCEDURE HEAD> <PROCEDURE NAME> ;
S                 CALL ENTER_PROC(VAR(MP),VAL(MP),-1,FIXEDTYPE);
R   <PROCEDURE NAME> <TYPE> ;
S                 CALL ENTER_PROC(VAR(MP),VAL(MP),-1,TYPE(SP-1));
R   <PROCEDURE NAME> ( <IDENTIFIER LIST> ) ;
S                 CALL ENTER_PROC(VAR(MP),VAL(MP),ID_LIST_PTR,
S                    FIXEDTYPE);
R   <PROCEDURE NAME> ( <IDENTIFIER LIST> ) <TYPE>;
S                 CALL ENTER_PROC(VAR(MP),VAL(MP),ID_LIST_PTR,
S                    TYPE(SP-1));
R <PROCEDURE NAME> <IDENTIFIER> % PROCEDURE
S                 VAL(MP) = PROC_DCL;
R   <IDENTIFIER> % RECURSIVE PROCEDURE
S                 VAL(MP) = RECURSIVE_PROC_DCL;
R <ENDING> END_PROC
S                 VAR(MP) = EMPTY_STRING_PTR;
E
P    SYN1
D           DECLARE(PROD,I,J,K)FIXED;
R   END_PROC <IDENTIFIER>
S                 VAR(MP) = VAR(SP);
```

```
R <BIT SPECIFICATION> <IDENTIFIER> ( <CONSTANT VALUE> )
S                 CALL CHECK_CONSTANT;
S                 IF VAL(MP) > 60 THENN
S                    CALL ERROR('PRECISION TOO BIG');
S                    VAL(SP) = 60;
S                 END;
S                 VAL(MP) = VAL(SP);
R <TYPE LIST> <TYPE>
S                 ID_LIST_PTR = 0;
S                 ID_TYPE(0) = TYPE(MP);
S                 ID_WIDTH(0) = VAL(MP);
R   <TYPE LIST> , <TYPE>
S                 INC_ID_PTR;
S                 ID_TYPE(ID_LIST_PTR)= TYPE(SP);
S                 ID_WIDTH(ID_LIST_PTR) = VAL(SP);
R <INITIAL LIST> INITIAL ( <CONSTANT LIST> )
R <CONSTANT LIST> <CONSTANT VALUE>
S                 IF LEVELNO > 0 THENN
S                    CALL ERROR('INITIAL VALUES ILLEGAL IN RECURSIVE PROCEDURE');
S                 END;
S                 PP = INITIAL_START;
S                 CALL SETINIT(MP);
R   <CONSTANT LIST> , <CONSTANT VALUE>
S                 CALL SETINIT(SP);
R <CONSTANT VALUE> <EXPRESSION>
S                 IF CLASS(MP) NE CONSTANT_CLASS THENN
S                    CALL ERROR('ONLY CONSTANT EXPRESSIONS ALLOWED');
S                    VAL(MP) = 0;
S                    VAR(MP) = EMPTY_STRING_PTR;
S                 END;
R <PROCEDURE DEFINITION> <PROCEDURE HEAD> <MAIN BODY> <ENDING>
S                 CALL CLOSE_SCOPE;
R <PROCEDURE HEAD> <PROCEDURE NAME> ;
S                 CALL ENTER_PROC(VAR(MP),VAL(MP),-1,FIXEDTYPE);
R   <PROCEDURE NAME> <TYPE> ;
S                 CALL ENTER_PROC(VAR(MP),VAL(MP),-1,TYPE(SP-1));
R   <PROCEDURE NAME> ( <IDENTIFIER LIST> ) ;
S                 CALL ENTER_PROC(VAR(MP),VAL(MP),ID_LIST_PTR,
S                    FIXEDTYPE);
R   <PROCEDURE NAME> ( <IDENTIFIER LIST> ) <TYPE>;
S                 CALL ENTER_PROC(VAR(MP),VAL(MP),ID_LIST_PTR,
S                    TYPE(SP-1));
R <PROCEDURE NAME> <IDENTIFIER> % PROCEDURE
S                 VAL(MP) = PROC_DCL;
R   <IDENTIFIER> % RECURSIVE PROCEDURE
S                 VAL(MP) = RECURSIVE_PROC_DCL;
R <ENDING> END_PROC
S                 VAR(MP) = EMPTY_STRING_PTR;
E
P    SYN1
D          DECLARE(PROD,I,J,K)FIXED;
R   END_PROC <IDENTIFIER>
S                 VAR(MP) = VAR(SP);
```

```
S                       DO I = 0 TO ID_LIST_PTR;
S                          IF OFFSET + ID_WIDTH(I) > 60 THENN
S                             OFFSET = 0; /* ALLOCATE NEW BLOCK */
S                             INITIAL_PP = PP;
S                             INITIAL_DP1 = DP1;
S                             PARALLEL_ALLOCATED = FALSE;
S                          ELSEE
S                             PP = INITIAL_PP;
S                             DP1 = INITIAL_DP1;
S                          END;
S                          CALL ENTER(ID_STACK(I),ARRAY_DCL,TYPE(SP),VAL(SP-1),
S                             OFFSET,ID_WIDTH(I));
S                          PARALLEL_ALLOCATED = TRUE;
S                          OFFSET = OFFSET + ID_WIDTH(I);
S                       END;
S                       DP1_LIMIT = DP1;
S                       INITIAL_LIMIT = PP; /* UPPER LIMIT FOR INITIAL VALUES */
S                    ELSEE
S                       CALL ERROR('FIXED TYPE REQUIRED');
S                    END;
S                    INITIAL_PRECISION = VAL(SP); /* NUMBER OF BITS PER ELEMENT */
S                    PARALLEL_ALLOCATED = FALSE;
R   <IDENTIFIER> ENTRY RETURNS ( <TYPE> )
S                    CALL ENTER_FORWARD_DCL(-1);
R   <IDENTIFIER> ENTRY ( <TYPE LIST> ) RETURNS ( <TYPE> )
S                    CALL ENTER_FORWARD_DCL(ID_LIST_PTR);
R <TYPE> FIXED
S                    TYPE(MP) = FIXEDTYPE;
S                    VAL(MP) = 60;
R   CHARACTER ( <CONSTANT VALUE> )
S                    TYPE(MP) = CHRTYPE;
S                    CALL CHECK_CONSTANT;
R   BIT ( <CONSTANT VALUE> )
S                    TYPE(MP) = CHRTYPE;
S                    CALL CHECK_CONSTANT;
R <DIMENSION> ( <CONSTANT VALUE> )
S                    CALL CHECK_CONSTANT;
R <IDENTIFIER LIST> <IDENTIFIER>
S                    ID_LIST_PTR = 0;
S                    ID_STACK(0) = VAR(MP);
R   <IDENTIFIER LIST> , <IDENTIFIER>
S                    INC_ID_PTR;
S                    ID_STACK(ID_LIST_PTR) = VAR(SP);
R <BIT FIELD LIST> <BIT SPECIFICATION>
S                    ID_LIST_PTR = 0;
S                    ID_STACK(0) = VAR(MP);
S                    ID_WIDTH(0) = VAL(MP);
R   <BIT FIELD LIST> , <BIT SPECIFICATION>
S                    INC_ID_PTR;
S                    ID_STACK(ID_LIST_PTR) = VAR(SP);
S                    ID_WIDTH(ID_LIST_PTR) = VAL(SP);
```

```
S                  DO I = 0 TO ID_LIST_PTR;
S                     IF OFFSET + ID_WIDTH(I) > 60 THENN
S                        OFFSET = 0; /* ALLOCATE NEW BLOCK */
S                        INITIAL_PP = PP;
S                        INITIAL_DP1 = DP1;
S                        PARALLEL_ALLOCATED = FALSE;
S                     ELSEE
S                        PP = INITIAL_PP;
S                        DP1 = INITIAL_DP1;
S                     END;
S                     CALL ENTER(ID_STACK(I),ARRAY_DCL,TYPE(SP),VAL(SP-1),
S                        OFFSET,ID_WIDTH(I));
S                     PARALLEL_ALLOCATED = TRUE;
S                     OFFSET = OFFSET + ID_WIDTH(I);
S                  END;
S                  DP1_LIMIT = DP1;
S                  INITIAL_LIMIT = PP; /* UPPER LIMIT FOR INITIAL VALUES */
S               ELSEE
S                  CALL ERROR('FIXED TYPE REQUIRED');
S               END;
S               INITIAL_PRECISION = VAL(SP); /* NUMBER OF BITS PER ELEMENT */
S               PARALLEL_ALLOCATED = FALSE;
R  <IDENTIFIER> ENTRY RETURNS ( <TYPE> )
S               CALL ENTER_FORWARD_DCL(-1);
R  <IDENTIFIER> ENTRY ( <TYPE LIST> ) RETURNS ( <TYPE> )
S               CALL ENTER_FORWARD_DCL(ID_LIST_PTR);
R <TYPE> FIXED
S               TYPE(MP) = FIXEDTYPE;
S               VAL(MP) = 60;
R  CHARACTER ( <CONSTANT VALUE> )
S               TYPE(MP) = CHRTYPE;
S               CALL CHECK_CONSTANT;
R  BIT ( <CONSTANT VALUE> )
S               TYPE(MP) = CHRTYPE;
S               CALL CHECK_CONSTANT;
R <DIMENSION> ( <CONSTANT VALUE> )
S               CALL CHECK_CONSTANT;
R <IDENTIFIER LIST> <IDENTIFIER>
S               ID_LIST_PTR = 0;
S               ID_STACK(0) = VAR(MP);
R  <IDENTIFIER LIST> , <IDENTIFIER>
S               INC_ID_PTR;
S               ID_STACK(ID_LIST_PTR) = VAR(SP);
R <BIT FIELD LIST> <BIT SPECIFICATION>
S               ID_LIST_PTR = 0;
S               ID_STACK(0) = VAR(MP);
S               ID_WIDTH(0) = VAL(MP);
R  <BIT FIELD LIST> , <BIT SPECIFICATION>
S               INC_ID_PTR;
S               ID_STACK(ID_LIST_PTR) = VAR(SP);
S               ID_WIDTH(ID_LIST_PTR) = VAL(SP);
```

```
R <EXPRESSION LIST> <EXPRESSION>
S                 FIXL(MP) = EXP_CNT;
S                 INC_EXP_CNT;
S                 CNT(MP) = 1;
S                 CLASS_STACK(EXP_CNT) = SHL(CLASS(MP),30) OR TYPE(MP);
S                 IF CLASS(MP) = CONSTANT_CLASS THENN
S                    IF TYPE(MP) = FIXEDTYPE THENN
S                       EXP_STACK(EXP_CNT) = VAL(MP);
S                    ELSEE
S                       EXP_STACK(EXP_CNT) = VAR(MP);
S                    END;
S                 ELSEE
S                    EXP_STACK(EXP_CNT) = G1(MP);
S                 END;
R   <EXPRESSION LIST> , <EXPRESSION>
S                 INC_EXP_CNT;
S                 CNT(MP) = CNT(MP) + 1;
S                 CLASS_STACK(EXP_CNT) = SHL(CLASS(SP),30) OR TYPE(SP);
S                 IF CLASS(SP) = CONSTANT_CLASS THENN
S                    IF TYPE(SP) = FIXEDTYPE THENN
S                       EXP_STACK(EXP_CNT) = VAL(SP);
S                    ELSEE
S                       EXP_STACK(EXP_CNT) = VAR(SP);
S                    END;
S                 ELSEE
S                    EXP_STACK(EXP_CNT) = G1(SP);   /* SAVE POINTER */
S                 END;
R <CONSTANT> <STRING>
S                 TYPE(MP) = CHRTYPE;
R   <NUMBER>
S                 TYPE(MP) = FIXEDTYPE;
```

```
START WRITING THE SEMANTIC PROCEDURES.
SEMANTIC PROCEDURES WRITTEN ONTO FILE OPUT2
THE GRAMMAR HAS BEEN READ.
```

	TERMINAL SYMBOLS	NONTERMINALS
1	_|_	<PROGRAM>
2	;	<MAIN BODY>
3	DECLARE	<DECLARATION LIST>
4	,	<STATEMENT LIST>
5	<IDENTIFIER>	<DECLARATION>
6	LITERALLY	<DECLARATION STATEMENT>
7	<STRING>	<PROCEDURE DEFINITION>
8	(	<DECLARATION ELEMENT LIST>
9	)	<DECLARATION ELEMENT>
10	ENTRY	<IDENTIFIER SPECIFICATION>
11	RETURNS	<INITIAL LIST>
12	FIXED	<TYPE>
13	CHARACTER	<DIMENSION>
14	BIT	<IDENTIFIER LIST>
15	INITIAL	<BIT FIELD LIST>
16	$	<TYPE LIST>
17	PROCEDURE	<CONSTANT VALUE>
18	RECURSIVE	<BIT SPECIFICATION>
19	END_PROC	<CONSTANT LIST>
20	EXIT	<EXPRESSION>
21	IF	<PROCEDURE HEAD>
22	THEN	<ENDING>
23	ELSE	<PROCEDURE NAME>
24	=	<STATEMENT>
25	OUTPUT	<BASIC STATEMENT>
26	FILE	<IF STATEMENT>
27	DO	<ASSIGNMENT>
28	END	<GROUP>
29	WHILE	<RETURN STATEMENT>
30	TO	<CALL STATEMENT>
31	CASE	<GENERATE BLOCK>
32	RETURN	<IF CLAUSE>
33	CALL	<TRUE PART>
34	GENERATE	<EXPRESSION LIST>
35	INLINE	<CASE BODY>
36	OR	<GENERATE HEAD>
37	AND	<INLINE BODY>
38	NOT	<LABEL>
39	<	<INLINE STATEMENT>
40	>	<LOGICAL FACTOR>
41	NE	<LOGICAL SECONDARY>
42	|	<LOGICAL PRIMARY>
43	+	<STRING EXPRESSION>
44	-	<RELATION>
45	*	<ARITHMETIC EXPRESSION>
46	/	<TERM>
47	MOD	<ADD>
48	<BUILTIN FUNCTION>	<PRIMARY>
49	<NUMBER>	<MULT>
50		<CONSTANT>
51		<VARIABLE>

	TERMINAL SYMBOLS	NONTERMINALS
1	_¦_	<PROGRAM>
2	;	<MAIN BODY>
3	DECLARE	<DECLARATION LIST>
4	,	<STATEMENT LIST>
5	<IDENTIFIER>	<DECLARATION>
6	LITERALLY	<DECLARATION STATEMENT>
7	<STRING>	<PROCEDURE DEFINITION>
8	(	<DECLARATION ELEMENT LIST>
9	)	<DECLARATION ELEMENT>
10	ENTRY	<IDENTIFIER SPECIFICATION>
11	RETURNS	<INITIAL LIST>
12	FIXED	<TYPE>
13	CHARACTER	<DIMENSION>
14	BIT	<IDENTIFIER LIST>
15	INITIAL	<BIT FIELD LIST>
16	$	<TYPE LIST>
17	PROCEDURE	<CONSTANT VALUE>
18	RECURSIVE	<BIT SPECIFICATION>
19	END_PROC	<CONSTANT LIST>
20	EXIT	<EXPRESSION>
21	IF	<PROCEDURE HEAD>
22	THEN	<ENDING>
23	ELSE	<PROCEDURE NAME>
24	=	<STATEMENT>
25	OUTPUT	<BASIC STATEMENT>
26	FILE	<IF STATEMENT>
27	DO	<ASSIGNMENT>
28	END	<GROUP>
29	WHILE	<RETURN STATEMENT>
30	TO	<CALL STATEMENT>
31	CASE	<GENERATE BLOCK>
32	RETURN	<IF CLAUSE>
33	CALL	<TRUE PART>
34	GENERATE	<EXPRESSION LIST>
35	INLINE	<CASE BODY>
36	OR	<GENERATE HEAD>
37	AND	<INLINE BODY>
38	NOT	<LABEL>
39	<	<INLINE STATEMENT>
40	>	<LOGICAL FACTOR>
41	NE	<LOGICAL SECONDARY>
42	¦	<LOGICAL PRIMARY>
43	+	<STRING EXPRESSION>
44	-	<RELATION>
45	*	<ARITHMETIC EXPRESSION>
46	/	<TERM>
47	MOD	<ADD>
48	<BUILTIN FUNCTION>	<PRIMARY>
49	<NUMBER>	<MULT>
50		<CONSTANT>
51		<VARIABLE>

THE PRODUCTIONS

```
 1  <PROGRAM>  ::=  <MAIN BODY>
 2  <MAIN BODY>  ::=  <DECLARATION LIST> <STATEMENT LIST>
 3                 |  <DECLARATION LIST>
 4                 |  <STATEMENT LIST>
 5  <DECLARATION LIST>  ::=  <DECLARATION>
 6                        |  <DECLARATION LIST> <DECLARATION>
 7  <DECLARATION>  ::=  <DECLARATION STATEMENT> ;
 8                   |  <PROCEDURE DEFINITION> ;
 9  <DECLARATION STATEMENT>  ::=  DECLARE <DECLARATION ELEMENT LIST>
10  <DECLARATION ELEMENT LIST>  ::=  <DECLARATION ELEMENT>
11                                |  <DECLARATION ELEMENT LIST> , <DECLARATION ELEMENT>
12  <DECLARATION ELEMENT>  ::=  <IDENTIFIER> LITERALLY <STRING>
13                           |  <IDENTIFIER SPECIFICATION>
14                           |  <IDENTIFIER SPECIFICATION> <INITIAL LIST>
15  <IDENTIFIER SPECIFICATION>  ::=  <IDENTIFIER> <TYPE>
16                                |  <IDENTIFIER> <DIMENSION> <TYPE>
17                                |  ( <IDENTIFIER LIST> ) <TYPE>
18                                |  ( <IDENTIFIER LIST> ) <DIMENSION> <TYPE>
19                                |  ( <BIT FIELD LIST> ) <DIMENSION> <TYPE>
20                                |  <IDENTIFIER> ENTRY RETURNS ( <TYPE> )
21                                |  <IDENTIFIER> ENTRY ( <TYPE LIST> ) RETURNS ( <TYPE> )
22  <TYPE>  ::=  FIXED
23            |  CHARACTER ( <CONSTANT VALUE> )
24            |  BIT ( <CONSTANT VALUE> )
25  <DIMENSION>  ::=  ( <CONSTANT VALUE> )
26  <IDENTIFIER LIST>  ::=  <IDENTIFIER>
27                       |  <IDENTIFIER LIST> , <IDENTIFIER>
28  <BIT FIELD LIST>  ::=  <BIT SPECIFICATION>
29                     |  <BIT FIELD LIST> , <BIT SPECIFICATION>
30  <BIT SPECIFICATION>  ::=  <IDENTIFIER> ( <CONSTANT VALUE> )
31  <TYPE LIST>  ::=  <TYPE>
32                |  <TYPE LIST> , <TYPE>
33  <INITIAL LIST>  ::=  INITIAL ( <CONSTANT LIST> )
34  <CONSTANT LIST>  ::=  <CONSTANT VALUE>
35                     |  <CONSTANT LIST> , <CONSTANT VALUE>
36  <CONSTANT VALUE>  ::=  <EXPRESSION>
37  <PROCEDURE DEFINITION>  ::=  <PROCEDURE HEAD> <MAIN BODY> <ENDING>
38  <PROCEDURE HEAD>  ::=  <PROCEDURE NAME> ;
39                      |  <PROCEDURE NAME> <TYPE> ;
40                      |  <PROCEDURE NAME> ( <IDENTIFIER LIST> ) ;
41                      |  <PROCEDURE NAME> ( <IDENTIFIER LIST> ) <TYPE> ;
42  <PROCEDURE NAME>  ::=  <IDENTIFIER> : PROCEDURE
43                      |  <IDENTIFIER> : RECURSIVE PROCEDURE
44  <ENDING>  ::=  END_PROC
45              |  END_PROC <IDENTIFIER>
46  <STATEMENT LIST>  ::=  <STATEMENT>
47                      |  <STATEMENT LIST> <STATEMENT>
48  <STATEMENT>  ::=  <BASIC STATEMENT>
49                 |  <IF STATEMENT>
```

```
 50  <BASIC STATEMENT>  ::=  <ASSIGNMENT> ;
 51                       |   <GROUP> ;
 52                       |   <RETURN STATEMENT> ;
 53                       |   <CALL STATEMENT> ;
 54                       |   EXIT ;
 55                       |   <GENERATE BLOCK> ;
 56                       |   ;
 57  <IF STATEMENT>  ::=  <IF CLAUSE> <STATEMENT>
 58                    |   <IF CLAUSE> <TRUE PART> <STATEMENT>
 59  <IF CLAUSE>  ::=  IF <EXPRESSION> THEN
 60  <TRUE PART>  ::=  <BASIC STATEMENT> ELSE
 61  <ASSIGNMENT>  ::=  <IDENTIFIER> = <EXPRESSION>
 62                  |   <IDENTIFIER> ( <EXPRESSION> ) = <EXPRESSION>
 63                  |   OUTPUT = <EXPRESSION>
 64                  |   OUTPUT ( <EXPRESSION> ) = <EXPRESSION>
 65                  |   FILE ( <EXPRESSION LIST> ) = <IDENTIFIER>
 66                  |   FILE ( <EXPRESSION LIST> ) = <IDENTIFIER> ( <EXPRESSION> )
 67                  |   <IDENTIFIER> = FILE ( <EXPRESSION LIST> )
 68                  |   <IDENTIFIER> ( <EXPRESSION> ) = FILE ( <EXPRESSION LIST> )
 69  <GROUP>  ::=  DO ; <STATEMENT LIST> END
 70           |   DO WHILE <EXPRESSION> ; <STATEMENT LIST> END
 71           |   DO <IDENTIFIER> = <EXPRESSION> TO <EXPRESSION> ; <STATEMENT LIST> END
 72           |   DO CASE <EXPRESSION> ; <CASE BODY> END
 73  <CASE BODY>  ::=  <STATEMENT>
 74               |   <CASE BODY> <STATEMENT>
 75  <RETURN STATEMENT>  ::=  RETURN
 76                       |   RETURN <EXPRESSION>
 77  <CALL STATEMENT>  ::=  CALL <IDENTIFIER>
 78                     |   CALL <IDENTIFIER> ( <EXPRESSION LIST> )
 79                     |   CALL EXIT
 80  <GENERATE BLOCK>  ::=  <GENERATE HEAD> <INLINE BODY> END
 81                     |   <GENERATE HEAD> <INLINE BODY> <LABEL> END
 82  <GENERATE HEAD>  ::=  GENERATE ;
 83  <INLINE BODY>  ::=  <INLINE STATEMENT>
 84                 |   <INLINE BODY> <INLINE STATEMENT>
 85  <INLINE STATEMENT>  ::=  INLINE ( <EXPRESSION LIST> ) ;
 86                       |   <LABEL> INLINE ( <EXPRESSION LIST> ) ;
 87  <LABEL>  ::=  <IDENTIFIER> :
 88  <EXPRESSION>  ::=  <LOGICAL FACTOR>
 89                 |   <EXPRESSION> OR <LOGICAL FACTOR>
 90  <LOGICAL FACTOR>  ::=  <LOGICAL SECONDARY>
 91                     |   <LOGICAL FACTOR> AND <LOGICAL SECONDARY>
 92  <LOGICAL SECONDARY>  ::=  <LOGICAL PRIMARY>
 93                        |   NOT <LOGICAL PRIMARY>
 94  <LOGICAL PRIMARY>  ::=  <STRING EXPRESSION>
 95                      |   <STRING EXPRESSION> <RELATION> <STRING EXPRESSION>
 96  <RELATION>  ::=  =
 97              |   <
 98              |   >
 99              |   < =
100              |   > =
101              |   NE
102  <STRING EXPRESSION>  ::=  <ARITHMETIC EXPRESSION>
103                        |   <STRING EXPRESSION> | | <ARITHMETIC EXPRESSION>
```

```
104   <ARITHMETIC EXPRESSION>  $$=  <TERM>
105                              !   <ARITHMETIC EXPRESSION> <ADD> <TERM>
106                              !   <ADD> <TERM>
107   <ADD>  $$=  +
108          !    -
109   <TERM>  $$=  <PRIMARY>
110           !    <TERM> <MULT> <PRIMARY>
111   <MULT>  $$=  *
112           !    /
113           !    MOD
114   <PRIMARY>  $$=  <CONSTANT>
115              !    <VARIABLE>
116              !    ( <EXPRESSION> )
117   <VARIABLE>  $$=  <IDENTIFIER>
118               !    <IDENTIFIER> ( <EXPRESSION LIST> )
119               !    <BUILTIN FUNCTION>
120               !    <BUILTIN FUNCTION> ( <EXPRESSION LIST> )
121   <EXPRESSION LIST>  $$=  <EXPRESSION>
122                      !    <EXPRESSION LIST> , <EXPRESSION>
123   <CONSTANT>  $$=  <STRING>
124               !    <NUMBER>
```

```
104   <ARITHMETIC EXPRESSION>  ::=  <TERM>
105                              |  <ARITHMETIC EXPRESSION> <ADD> <TERM>
106                              |  <ADD> <TERM>
107   <ADD>  ::=  +
108           |   -
109   <TERM>  ::=  <PRIMARY>
110            |   <TERM> <MULT> <PRIMARY>
111   <MULT>  ::=  *
112            |   /
113            |   MOD
114   <PRIMARY>  ::=  <CONSTANT>
115               |   <VARIABLE>
116               |   ( <EXPRESSION> )
117   <VARIABLE>  ::=  <IDENTIFIER>
118                |   <IDENTIFIER> ( <EXPRESSION LIST> )
119                |   <BUILTIN FUNCTION>
120                |   <BUILTIN FUNCTION> ( <EXPRESSION LIST> )
121   <EXPRESSION LIST>  ::=  <EXPRESSION>
122                       |   <EXPRESSION LIST> , <EXPRESSION>
123   <CONSTANT>  ::=  <STRING>
124                |   <NUMBER>
```

```
SOME STATISTICS ON THE GRAMMARS

    NUMBER OF TERMINAL SYMBOLS = 49
    NUMBER OF NONTERMINAL SYMBOLS = 51
         TOTAL NUMBER OF SYMBOLS = 100
    NUMBER OF PRODUCTIONS = 124
    SPACE REQUIRED TO STORE THE PRODUCTIONS = 1117 BYTES NOT INCLUDING THE VOCABULARY.
    THE AVERAGE LENGTH OF THE RIGHT PARTS OF PRODUCTIONS = 2.65 SYMBOLS.
```

```
START CONSTRUCTING THE GRAMMAR'S CHARACTERISTIC FSM.
THE CFSM FOR THE GRAMMAR HAS BEEN COMPUTED.
THE GRAMMAR IS NOT LR(0).  LOOK-AHEAD MUST BE ADDED.


START COMPUTING LOOK-AHEAD SETS.
LOOK-AHEAD SETS HAVE BEEN COMPUTED.
THE GRAMMAR IS SLR(1).
```

THE LOOK-AHEAD SETS

NONTERMINAL	TERMINAL SYMBOLS																																																
SYMBOLS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
<PROGRAM>	1																																																
<MAIN BODY>	1																		1																														
<DECLARATIO	1	1	1		1														1	1	1				1	1	1					1	1	1															
<STATEMENT	1	1			1														1	1	1				1	1	1	1				1	1	1															
<DECLARATIO	1	1	1		1														1	1	1				1	1	1					1	1	1															
<DECLARATIO		1																																															
<PROCEDURE		1																																															
<DECLARATIO		1		1																																													
<DECLARATIO		1		1																																													
<IDENTIFIER		1		1											1																																		
<INITIAL LI		1		1																																													
<TYPE>		1		1					1						1																																		
<DIMENSION>												1	1	1																																		
<IDENTIFIER				1					1																																								
<BIT FIELD				1					1																																								
<TYPE LIST>				1					1																																								
<CONSTANT V				1					1																																								
<BIT SPECIF				1					1																																								
<CONSTANT L				1					1																																								
<EXPRESSION		1		1					1													1								1						1													
<PROCEDURE		1	1		1															1	1				1	1	1					1	1	1															
<ENDING>		1																																															
<PROCEDURE		1						1				1	1	1																																			
<STATEMENT>	1	1			1														1	1	1				1	1	1	1				1	1	1															
<BASIC STAT	1	1			1														1	1	1		1		1	1	1	1				1	1	1															
<IF STATEME	1	1			1														1	1	1				1	1	1	1				1	1	1															
<ASSIGNMENT		1																																															
<GROUP>		1																																															
<RETURN STA		1																																															
<CALL STATE		1																																															
<GENERATE B		1																																															
<IF CLAUSE>		1			1															1	1				1	1	1					1	1	1															
<TRUE PART>		1			1															1	1				1	1	1					1	1	1															
<EXPRESSION				1					1																																								
<CASE BODY>		1			1															1	1				1	1	1	1				1	1	1															
<GENERATE H					1																														1														
<INLINE BOD					1																							1							1														
<LABEL>																												1							1														
<INLINE STA					1																							1							1														
<LOGICAL FA		1		1					1													1								1						1	1												
<LOGICAL SE		1		1					1													1								1						1	1												
<LOGICAL PR		1		1					1													1								1						1	1												
<STRING EXP		1		1					1													1		1						1						1	1		1	1	1	1							
<RELATION>					1		1	1																																			1	1				1	1
<ARITHMETIC		1		1					1													1		1						1						1	1		1	1	1	1	1	1					
<TERM>		1		1					1													1		1						1						1	1		1	1	1	1	1	1	1	1	1		
<ADD>					1		1	1																																								1	1
<PRIMARY>		1		1					1													1		1						1						1	1		1	1	1	1	1	1	1	1	1		
<MULT>					1		1	1																																								1	1
<CONSTANT>		1		1					1													1		1						1						1	1		1	1	1	1	1	1	1	1	1		
<VARIABLE>		1		1					1													1		1						1						1	1		1	1	1	1	1	1	1	1	1		

THE LOOK-AHEAD SETS

NONTERMINAL SYMBOLS	TERMINAL SYMBOLS																																																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
<PROGRAM>	1																																																
<MAIN BODY>	1																		1																														
<DECLARATIO	1	1	1		1														1	1	1				1	1	1					1	1	1															
<STATEMENT	1	1			1														1	1	1				1	1	1	1				1	1	1															
<DECLARATIO	1	1	1		1														1	1	1				1	1	1					1	1	1															
<DECLARATIO		1																																															
<PROCEDURE		1																																															
<DECLARATIO		1		1																																													
<DECLARATIO		1		1																																													
<IDENTIFIER		1		1											1																																		
<INITIAL LI		1		1																																													
<TYPE>		1		1					1						1																																		
<DIMENSION>												1	1	1																																			
<IDENTIFIER				1					1																																								
<BIT FIELD				1					1																																								
<TYPE LIST>				1					1																																								
<CONSTANT V				1					1																																								
<BIT SPECIF				1					1																																								
<CONSTANT L				1					1																																								
<EXPRESSION		1		1					1													1								1						1													
<PROCEDURE		1	1		1															1	1				1	1	1					1	1	1															
<ENDING>		1																																														
<PROCEDURE		1						1				1	1	1																																		
<STATEMENT>	1	1			1														1	1	1				1	1	1	1				1	1	1															
<BASIC STAT	1	1			1														1	1	1		1		1	1	1	1				1	1	1															
<IF STATEME	1	1			1														1	1	1				1	1	1	1				1	1	1															
<ASSIGNMENT		1																																														
<GROUP>		1																																														
<RETURN STA		1																																														
<CALL STATE		1																																														
<GENERATE B		1																																														
<IF CLAUSE>		1			1															1	1				1	1	1					1	1	1															
<TRUE PART>		1			1															1	1				1	1	1					1	1	1															
<EXPRESSION				1					1																																							
<CASE BODY>		1			1															1	1				1	1	1	1				1	1	1															
<GENERATE H					1																														1														
<INLINE BOD					1																							1							1														
<LABEL>																												1							1														
<INLINE STA					1																							1							1														
<LOGICAL FA		1		1					1													1								1						1	1												
<LOGICAL SE		1		1					1													1								1						1	1												
<LOGICAL PR		1		1					1													1								1						1	1												
<STRING EXP		1		1					1													1		1						1						1	1		1	1	1	1							
<RELATION>					1		1	1																																			1	1				1	1
<ARITHMETIC		1		1					1													1		1						1						1	1		1	1	1	1	1	1					
<TERM>		1		1					1													1		1						1						1	1		1	1	1	1	1	1	1	1	1		
<ADD>					1		1	1																																							1	1
<PRIMARY>		1		1					1													1		1						1						1	1		1	1	1	1	1	1	1	1	1		
<MULT>					1		1	1																																							1	1
<CONSTANT>		1		1					1													1		1						1						1	1		1	1	1	1	1	1	1	1	1		
<VARIABLE>		1		1					1													1		1						1						1	1		1	1	1	1	1	1	1	1	1		

```
START WRITING THE ERROR RECOVERY TABLE.
ERROR RECOVERY TABLE WRITTEN ONTO FILE OPUT2
```

```
START WRITING THE ERROR RECOVERY TABLE.
ERROR RECOVERY TABLE WRITTEN ONTO FILE OPUT2
```

FOLLOWING ARE THE LEGAL SUCCESSORS OF TERMINAL SYMBOLS.

```
-----------------------------------------------------------------------------------------------------------
.
. _|_
. ---->  _|_  ;  DECLARE  <IDENTIFIER>  EXIT  IF  OUTPUT  FILE  DO  RETURN  CALL  GENERATE
.
-----------------------------------------------------------------------------------------------------------
.
. ;
. ---->  _|_  ;  DECLARE  <IDENTIFIER>  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE  DO  END  RETURN  CALL  GENERATE  INLINE
.
-----------------------------------------------------------------------------------------------------------
.
. DECLARE
. ---->  <IDENTIFIER>  (
.
-----------------------------------------------------------------------------------------------------------
.
. ,
. ---->  <IDENTIFIER>  <STRING>  (  FIXED  CHARACTER  BIT  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------
.
. <IDENTIFIER>
. ---->  ;  ,  LITERALLY  (  )  ENTRY  FIXED  CHARACTER  BIT  &  THEN  =  TO  OR  AND  <  >  NE  |  +  -  *  /  MOD
.
-----------------------------------------------------------------------------------------------------------
.
. LITERALLY
. ---->  <STRING>
.
-----------------------------------------------------------------------------------------------------------
.
. <STRING>
. ---->  ;  ,  )  THEN  =  TO  OR  AND  <  >  NE  |  +  -  *  /  MOD
.
-----------------------------------------------------------------------------------------------------------
.
. (
. ---->  <IDENTIFIER>  <STRING>  (  FIXED  CHARACTER  BIT  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------
.
. )
. ---->  ;  ,  (  )  RETURNS  FIXED  CHARACTER  BIT  INITIAL  THEN  =  TO  OR  AND  <  >  NE  |  +  -  *  /  MOD
.
-----------------------------------------------------------------------------------------------------------
```

FOLLOWING ARE THE LEGAL SUCCESSORS OF TERMINAL SYMBOLS.

```
--------------------------------------------------------------------------------------------------------
.
. _|_
. ---->  _|_  ;  DECLARE  <IDENTIFIER>  EXIT  IF  OUTPUT  FILE  DO  RETURN  CALL  GENERATE
.
--------------------------------------------------------------------------------------------------------
.
. ;
. ---->  _|_  ;  DECLARE  <IDENTIFIER>  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE  DO  END  RETURN  CALL  GENERATE  INLINE
.
--------------------------------------------------------------------------------------------------------
.
. DECLARE
.  ---->  <IDENTIFIER>  (
.
--------------------------------------------------------------------------------------------------------
.
. ,
. ---->  <IDENTIFIER>  <STRING>  (  FIXED  CHARACTER  BIT  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------------------------------
.
. <IDENTIFIER>
.  ---->  ;  ,  LITERALLY  (  )  ENTRY  FIXED  CHARACTER  BIT  $  THEN  =  TO  OR  AND  <  >  NE  |  +  -  *  /  MOD
.
--------------------------------------------------------------------------------------------------------
.
. LITERALLY
.  ---->  <STRING>
.
--------------------------------------------------------------------------------------------------------
.
. <STRING>
.  ---->  ;  ,  )  THEN  =  TO  OR  AND  <  >  NE  |  +  -  *  /  MOD
.
--------------------------------------------------------------------------------------------------------
.
. (
.  ---->  <IDENTIFIER>  <STRING>  (  FIXED  CHARACTER  BIT  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------------------------------
.
. )
.  ---->  ;  ,  (  )  RETURNS  FIXED  CHARACTER  BIT  INITIAL  THEN  =  TO  OR  AND  <  >  NE  |  +  -  *  /  MOD
.
--------------------------------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
.
. ENTRY
.  ---->  (  RETURNS
.
--------------------------------------------------------------------------------
.
. RETURNS
.  ---->  (
.
--------------------------------------------------------------------------------
.
. FIXED
.  ---->  ;  , )  INITIAL
.
--------------------------------------------------------------------------------
.
. CHARACTER
.  ---->  (
.
--------------------------------------------------------------------------------
.
. BIT
.  ---->  (
.
--------------------------------------------------------------------------------
.
. INITIAL
.  ---->  (
.
--------------------------------------------------------------------------------
.
. ;
.  ---->  PROCEDURE  RECURSIVE  END  INLINE
.
--------------------------------------------------------------------------------
.
. PROCEDURE
.  ---->  ;  (  FIXED  CHARACTER  BIT
.
--------------------------------------------------------------------------------
.
. RECURSIVE
.  ---->  PROCEDURE
.
--------------------------------------------------------------------------------
.
. END_PROC
.  ---->  ;  <IDENTIFIER>
.
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
. EXIT
.  ---->  ;
.
--------------------------------------------------------------------------------
.
. IF
.  ---->  <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. THEN
.  ---->  ;  <IDENTIFIER>  EXIT  IF  OUTPUT  FILE  DO  RETURN  CALL  GENERATE
.
--------------------------------------------------------------------------------
.
. ELSE
.  ---->  ;  <IDENTIFIER>  EXIT  IF  OUTPUT  FILE  DO  RETURN  CALL  GENERATE
.
--------------------------------------------------------------------------------
.
. =
.  ---->  <IDENTIFIER>  <STRING>  (  FILE  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. OUTPUT
.  ---->  (  =
.
--------------------------------------------------------------------------------
.
. FILE
.  ---->  (
.
--------------------------------------------------------------------------------
.
. DO
.  ---->  ;  <IDENTIFIER>  WHILE  CASE
.
--------------------------------------------------------------------------------
.
. END
.  ---->  ;
.
--------------------------------------------------------------------------------
.
. WHILE
.  ---->  <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. TO
.  ---->  <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
```

```
--------------------------------------------------------------------------------
.
. CASE
.  ----> <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. RETURN
.  ---->  ;  <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. CALL
.  ----> <IDENTIFIER>  EXIT
.
--------------------------------------------------------------------------------
.
. GENERATE
.  ---->  ;
.
--------------------------------------------------------------------------------
.
. INLINE
.  ---->  (
.
--------------------------------------------------------------------------------
.
. OR
.  ----> <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. AND
.  ----> <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. NOT
.  ----> <IDENTIFIER>  <STRING>  (  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. <
.  ----> <IDENTIFIER>  <STRING>  (  =  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
.
. >
.  ----> <IDENTIFIER>  <STRING>  (  =  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------
. CASE
.  ---->  <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------
.
. RETURN
.  ---->  ;  <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------
.
. CALL
.  ---->  <IDENTIFIER>  EXIT
.
------------------------------------------------------------------------------------------
.
. GENERATE
.  ---->  ;
.
------------------------------------------------------------------------------------------
.
. INLINE
.  ---->  (
.
------------------------------------------------------------------------------------------
.
. OR
.  ---->  <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------
.
. AND
.  ---->  <IDENTIFIER>  <STRING>  (  NOT  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------
.
. NOT
.  ---->  <IDENTIFIER>  <STRING>  (  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------
.
. <
.  ---->  <IDENTIFIER>  <STRING>  (  =  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------
.
. >
.  ---->  <IDENTIFIER>  <STRING>  (  =  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------
. NE
.  ---->  <IDENTIFIER>  <STRING>  (  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
---------------------------------------------------------------------------
.
. !
.  ---->  <IDENTIFIER>  <STRING>  (  !  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
---------------------------------------------------------------------------
.
. +
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
---------------------------------------------------------------------------
.
. -
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
---------------------------------------------------------------------------
.
. *
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
---------------------------------------------------------------------------
.
. /
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
---------------------------------------------------------------------------
.
. MOD
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
---------------------------------------------------------------------------
.
. <BUILTIN FUNCTION>
.  ---->  ;  ,  (  )  THEN  =  TO  OR  AND  <  >  NE  !  +  -  *  /  MOD
.
---------------------------------------------------------------------------
.
. <NUMBER>
.  ---->  ;  ,  )  THEN  =  TO  OR  AND  <  >  NE  !  +  -  *  /  MOD
.
---------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------------
.
. NE
.  ---->  <IDENTIFIER>  <STRING>  (  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------
.
. !
.  ---->  <IDENTIFIER>  <STRING>  (  !  +  -  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------
.
. +
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------
.
. -
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------
.
. *
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------
.
. /
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------
.
. MOD
.  ---->  <IDENTIFIER>  <STRING>  (  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------
.
. <BUILTIN FUNCTION>
.  ---->  ;  ,  (  )  THEN  =  TO  OR  AND  <  >  NE  !  +  -  *  /  MOD
.
-----------------------------------------------------------------------------------
.
. <NUMBER>
.  ---->  ;  ,  )  THEN  =  TO  OR  AND  <  >  NE  !  +  -  *  /  MOD
.
-----------------------------------------------------------------------------------
```

FOLLOWING ARE THE ILLEGAL SUCCESSORS OF TERMINAL SYMBOLS.

```
-------------------------------------------------------------------------------------------------------------------
.
. _!_
.  ---->  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC  THEN
.  ---->  ELSE  =  END  WHILE  TO  CASE  INLINE  OR  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
. ;
.  ---->  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  THEN  =  WHILE
.  ---->  TO  CASE  OR  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. DECLARE
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT
.  ---->  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. ,
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =
.  ---->  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
.
. <IDENTIFIER>
.  ---->  _!_  DECLARE  <IDENTIFIER>  <STRING>  RETURNS  INITIAL  PROCEDURE  RECURSIVE  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE  DO
.  ---->  END  WHILE  CASE  RETURN  CALL  GENERATE  INLINE  NOT  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. LITERALLY
.  ---->  _!_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. <STRING>
.  ---->  _!_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  (  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE  DO  END  WHILE  CASE  RETURN  CALL  GENERATE  INLINE  NOT
.  ---->  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
```

```
FOLLOWING ARE THE ILLEGAL SUCCESSORS OF TERMINAL SYMBOLS.

-----------------------------------------------------------------------------------------------------------------
.
. _!_
.  ---->  , LITERALLY <STRING> ( ) ENTRY RETURNS FIXED CHARACTER BIT INITIAL $ PROCEDURE RECURSIVE END_PROC THEN
.  ---->  ELSE = END WHILE TO CASE INLINE OR AND NOT < > NE ! + - * / MOD <BUILTIN_FUNCTION> <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------
.
. ;
.  ---->  , LITERALLY <STRING> ( ) ENTRY RETURNS FIXED CHARACTER BIT INITIAL $ PROCEDURE RECURSIVE THEN = WHILE
.  ---->  TO CASE OR AND NOT < > NE ! + - * / MOD <BUILTIN FUNCTION> <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------
. DECLARE
.  ---->  _!_ ; DECLARE , LITERALLY <STRING> ) ENTRY RETURNS FIXED CHARACTER BIT INITIAL $ PROCEDURE RECURSIVE
.  ---->  END_PROC EXIT IF THEN ELSE = OUTPUT FILE DO END WHILE TO CASE RETURN CALL GENERATE INLINE OR AND NOT
.  ---->  < > NE ! + - * / MOD <BUILTIN FUNCTION> <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------
.
. ,
.  ---->  _!_ ; DECLARE , LITERALLY ) ENTRY RETURNS INITIAL $ PROCEDURE RECURSIVE END_PROC EXIT IF THEN ELSE =
.  ---->  OUTPUT FILE DO END WHILE TO CASE RETURN CALL GENERATE INLINE OR AND < > NE ! * / MOD
.
-----------------------------------------------------------------------------------------------------------------
.
. <IDENTIFIER>
.  ---->  _!_ DECLARE <IDENTIFIER> <STRING> RETURNS INITIAL PROCEDURE RECURSIVE END_PROC EXIT IF ELSE OUTPUT FILE DO
.  ---->  END WHILE CASE RETURN CALL GENERATE INLINE NOT <BUILTIN FUNCTION> <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------
.
. LITERALLY
.  ---->  _!_ ; DECLARE , <IDENTIFIER> LITERALLY ( ) ENTRY RETURNS FIXED CHARACTER BIT INITIAL $ PROCEDURE
.  ---->  RECURSIVE END_PROC EXIT IF THEN ELSE = OUTPUT FILE DO END WHILE TO CASE RETURN CALL GENERATE INLINE OR
.  ---->  AND NOT < > NE ! + - * / MOD <BUILTIN FUNCTION> <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------
.
. <STRING>
.  ---->  _!_ DECLARE <IDENTIFIER> LITERALLY <STRING> ( ENTRY RETURNS FIXED CHARACTER BIT INITIAL $ PROCEDURE
.  ---->  RECURSIVE END_PROC EXIT IF ELSE OUTPUT FILE DO END WHILE CASE RETURN CALL GENERATE INLINE NOT
.  ---->  <BUILTIN FUNCTION> <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------------------------------------
.
. (
.    ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  INITIAL  &  PROCEDURE  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =
.    ---->  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  ¬NE  !  *  /  MOD
.
----------------------------------------------------------------------------------------------------------
.
. )
.    ---->  _|_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  ENTRY  &  PROCEDURE  RECURSIVE  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE
.    ---->  DO  END  WHILE  CASE  RETURN  CALL  GENERATE  INLINE  NOT  <BUILTIN FUNCTION>  <NUMBER>
.
----------------------------------------------------------------------------------------------------------
.
. ENTRY
.    ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  FIXED  CHARACTER  BIT  INITIAL  &  PROCEDURE  RECURSIVE
.    ---->  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT
.    ---->  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
----------------------------------------------------------------------------------------------------------
.
. RETURNS
.    ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  &  PROCEDURE
.    ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.    ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
----------------------------------------------------------------------------------------------------------
.
. FIXED
.    ---->  _|_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  (  ENTRY  RETURNS  FIXED  CHARACTER  BIT  &  PROCEDURE  RECURSIVE
.    ---->  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT
.    ---->  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
----------------------------------------------------------------------------------------------------------
.
. CHARACTER
.    ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  &  PROCEDURE
.    ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.    ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
----------------------------------------------------------------------------------------------------------
.
. BIT
.    ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  &  PROCEDURE
.    ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.    ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
----------------------------------------------------------------------------------------------------------
.
. INITIAL
.    ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  &  PROCEDURE
.    ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.    ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
----------------------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------------------------
.
. (
.  ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =
.  ---->  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  |  *  /  MOD
.
------------------------------------------------------------------------------------------------------------
.
. )
.  ---->  _|_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  ENTRY  $  PROCEDURE  RECURSIVE  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE
.  ---->  DO  END  WHILE  CASE  RETURN  CALL  GENERATE  INLINE  NOT  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------------------------
.
. ENTRY
.  ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT
.  ---->  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------------------------
.
. RETURNS
.  ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------------------------
.
. FIXED
.  ---->  _|_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  (  ENTRY  RETURNS  FIXED  CHARACTER  BIT  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT
.  ---->  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------------------------
.
. CHARACTER
.  ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------------------------
.
. BIT
.  ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------------------------
.
. INITIAL
.  ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
------------------------------------------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------------------------------------------------
.
. $
.  ---->  _!_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  WHILE  TO  CASE  RETURN  CALL  GENERATE  OR  AND  NOT  <  >  NE  !  +  -  *
.  ---->  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------------
. PROCEDURE
.  ---->  _!_  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC  EXIT
.  ---->  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE  !  +
.  ---->  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------------
.
. RECURSIVE
.  ---->  _!_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
-----------------------------------------------------------------------------------------------------------------------
.
. END_PROC
.  ---->  _!_  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT
.  ---->  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------------
. EXIT
.  ---->  _!_  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------------
.
. IF
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.  ---->  /  MOD
.
-----------------------------------------------------------------------------------------------------------------------
.
. THEN
.  ---->  _!_  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  THEN  ELSE  =  END  WHILE  TO  CASE  INLINE  OR  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>
.  ---->  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------------
.
. ELSE
.  ---->  _!_  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  THEN  ELSE  =  END  WHILE  TO  CASE  INLINE  OR  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>
.  ---->  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------------------------
.
.
.  ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  WHILE  TO  CASE  RETURN  CALL  GENERATE  OR  AND  NOT  <  >  NE  |  +  -  *
.  ---->  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------
.
. PROCEDURE
.  ---->  _|_  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC  EXIT
.  ---->  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE  |  +
.  ---->  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------
.
. RECURSIVE
.  ---->  _|_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------
.
. END_PROC
.  ---->  _|_  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT
.  ---->  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------
.
. EXIT
.  ---->  _|_  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------
.
. IF
.  ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  |  *
.  ---->  /  MOD
.
-----------------------------------------------------------------------------------------------
.
. THEN
.  ---->  _|_  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  THEN  ELSE  =  END  WHILE  TO  CASE  INLINE  OR  AND  NOT  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>
.  ---->  <NUMBER>
.
-----------------------------------------------------------------------------------------------
.
. ELSE
.  ---->  _|_  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  THEN  ELSE  =  END  WHILE  TO  CASE  INLINE  OR  AND  NOT  <  >  NE  |  +  -  *  /  MOD  <BUILTIN FUNCTION>
.  ---->  <NUMBER>
.
-----------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------------------------------
.
. =
. ---->   !  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
. ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
. ---->  MOD
.
-------------------------------------------------------------------------------------------------------------------
.
. OUTPUT
. ---->   !  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
. ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
. ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. FILE
. ---->   !  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
. ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
. ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. DO
. ---->   !  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
. ---->  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  TO  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE  !
. ---->  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. END
. ---->   !  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  BROCEDURE
. ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
. ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. WHILE
. ---->   !  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
. ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
. ---->  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
.
. TO
. ---->   !  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
. ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
. ---->  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------------------------
.
. =
.   ---->   !  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.   ---->   EXIT  IF  THEN  ELSE  =  OUTPUT  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *  /
.   ---->   MOD
.
-------------------------------------------------------------------------------------------------------------
.
. OUTPUT
.   ---->   !  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.   ---->   RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.   ---->   AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------
.
. FILE
.   ---->   !  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.   ---->   RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.   ---->   AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------
.
. DO
.   ---->   !  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.   ---->   END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  TO  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE  !
.   ---->   +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------
.
. END
.   ---->   !  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.   ---->   RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.   ---->   AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------
.
. WHILE
.   ---->   !  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.   ---->   EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.   ---->   /  MOD
.
-------------------------------------------------------------------------------------------------------------
.
. TO
.   ---->   !  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.   ---->   EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.   ---->   /  MOD
.
-------------------------------------------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------------------------------------
.
. CASE
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.  ---->  /  MOD
.
-----------------------------------------------------------------------------------------------------------
.
. RETURN
.  ---->  _!_  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC  EXIT
.  ---->  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *  /
.  ---->  MOD
.
-----------------------------------------------------------------------------------------------------------
.
. CALL
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >
.  ---->  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------
.
. GENERATE
.  ---->  _!_  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------
.
. INLINE
.  ---->  _!_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-----------------------------------------------------------------------------------------------------------
.
. OR
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.  ---->  /  MOD
.
-----------------------------------------------------------------------------------------------------------
.
. AND
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.  ---->  /  MOD
.
-----------------------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------------------------------
.
. CASE
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.  ---->  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
.
. RETURN
.  ---->  _!_  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC  EXIT
.  ---->  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *  /
.  ---->  MOD
.
-------------------------------------------------------------------------------------------------------------------
.
. CALL
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >
.  ---->  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. GENERATE
.  ---->  _!_  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  (  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. INLINE
.  ---->  _!_  ;  DECLARE  ,  <IDENTIFIER>  LITERALLY  <STRING>  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR
.  ---->  AND  NOT  <  >  NE  !  +  -  *  /  MOD  <BUILTIN FUNCTION>  <NUMBER>
.
-------------------------------------------------------------------------------------------------------------------
.
. OR
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.  ---->  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
.
. AND
.  ---->  _!_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  <  >  NE  !  *
.  ---->  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------------------------------
. NOT
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
. <
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE  |
.  ---->  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
. >
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE  |
.  ---->  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
. NE
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
. |
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
. +
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
. -
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------------------------------------
. NOT
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------------
.
. <
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE  |
.  ---->  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------------
.
. >
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE  |
.  ---->  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------------
.
. NE
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------------
.
. |
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------------
.
. +
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------------
.
. -
.  ---->  |  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
-------------------------------------------------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------------------------------------------------------
.
. *
.  ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
--------------------------------------------------------------------------------------------------------------------------------
.
. /
.  ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
--------------------------------------------------------------------------------------------------------------------------------
.
. MOD
.  ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
--------------------------------------------------------------------------------------------------------------------------------
.
. <BUILTIN FUNCTION>
.  ---->  _|_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE  DO  END  WHILE  CASE  RETURN  CALL  GENERATE  INLINE  NOT  <BUILTIN FUNCTION>
.  ---->  <NUMBER>
.
--------------------------------------------------------------------------------------------------------------------------------
.
. <NUMBER>
.  ---->  _|_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  (  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE  DO  END  WHILE  CASE  RETURN  CALL  GENERATE  INLINE  NOT
.  ---->  <BUILTIN FUNCTION>  <NUMBER>
.
--------------------------------------------------------------------------------------------------------------------------------
```

```
.-------------------------------------------------------------------------------------------------------------
. *
.  ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
.-------------------------------------------------------------------------------------------------------------
. /
.  ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
.-------------------------------------------------------------------------------------------------------------
. MOD
.  ---->  _|_  ;  DECLARE  ,  LITERALLY  )  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE  END_PROC
.  ---->  EXIT  IF  THEN  ELSE  =  OUTPUT  FILE  DO  END  WHILE  TO  CASE  RETURN  CALL  GENERATE  INLINE  OR  AND  NOT  <  >  NE
.  ---->  |  +  -  *  /  MOD
.
.-------------------------------------------------------------------------------------------------------------
. <BUILTIN FUNCTION>
.  ---->  _|_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE  RECURSIVE
.  ---->  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE  DO  END  WHILE  CASE  RETURN  CALL  GENERATE  INLINE  NOT  <BUILTIN FUNCTION>
.  ---->  <NUMBER>
.
.-------------------------------------------------------------------------------------------------------------
. <NUMBER>
.  ---->  _|_  DECLARE  <IDENTIFIER>  LITERALLY  <STRING>  (  ENTRY  RETURNS  FIXED  CHARACTER  BIT  INITIAL  $  PROCEDURE
.  ---->  RECURSIVE  END_PROC  EXIT  IF  ELSE  OUTPUT  FILE  DO  END  WHILE  CASE  RETURN  CALL  GENERATE  INLINE  NOT
.  ---->  <BUILTIN FUNCTION>  <NUMBER>
.
.-------------------------------------------------------------------------------------------------------------
```

```
START COMPUTING THE GROUPING TABLE.
START WRITTING GROUPING TABLE
GROUPING TABLE WRITTEN ONTO FILE OPUT2
START COMPUTING ACTION TABLE.
ACTION TABLE HAS BEEN COMPUTED.
START WRITTING ACTION TABLE.
ACTION TABLE WRITTEN ONTO FILE OPUT2.
START COMPUTING THE DPDA.
THE DPDA HAS BEEN COMPUTED.
```

```
START COMPUTING THE GROUPING TABLE.
START WRITTING GROUPING TABLE
GROUPING TABLE WRITTEN ONTO FILE OPUT2
START COMPUTING ACTION TABLE.
ACTION TABLE HAS BEEN COMPUTED.
START WRITTING ACTION TABLE.
ACTION TABLE WRITTEN ONTO FILE OPUT2.
START COMPUTING THE DPDA.
THE DPDA HAS BEEN COMPUTED.
```

```
START WRITING THE PARSING TABLES.
TABLES WRITTEN ONTO FILE OPUT3
GRAMMAR IS ANALYSED.
```

```
START WRITING THE PARSING TABLES.
TABLES WRITTEN ONTO FILE OPUT3
GRAMMAR IS ANALYSED.
```

3) PARSING TABLES GENERATED BY THE SLR(1) LANGUAGE ANALYZER.

3) PARSING TABLES GENERATED BY THE SLR(1) LANGUAGE ANALYZER.

```
/*  THESE ARE SLR(1) PARSING TABLES  */

   DECLARE NO_TERMINALS LITERALLY '49',
           NO_NTS       LITERALLY '51',
           NO_SYMS      LITERALLY '100';

   DECLARE V(NO_SYMS) CHARACTER(26) INITIAL ('ERROR_SYMBOL', '_|_', ';'
      , 'DECLARE', ',', '<IDENTIFIER>', 'LITERALLY', '<STRING>', '(', ')'
      , 'ENTRY', 'RETURNS', 'FIXED', 'CHARACTER', 'BIT', 'INITIAL', '$'
      , 'PROCEDURE', 'RECURSIVE', 'END PROC', 'EXIT', 'IF', 'THEN', 'ELSE'
      , '=', 'OUTPUT', 'FILE', 'DO', 'END', 'WHILE', 'TO', 'CASE', 'RETURN'
      , 'CALL', 'GENERATE', 'INLINE', 'OR', 'AND', 'NOT', '<', '>', 'NE', '|'
      , '+', '-', '*', '/', 'MOD', '<BUILTIN FUNCTION>', '<NUMBER>'
      , '<PROGRAM>', '<MAIN BODY>', '<DECLARATION LIST>', '<STATEMENT LIST>'
      , '<DECLARATION>', '<DECLARATION STATEMENT>', '<PROCEDURE DEFINITION>'
      , '<DECLARATION ELEMENT LIST>', '<DECLARATION ELEMENT>'
      , '<IDENTIFIER SPECIFICATION>', '<INITIAL LIST>', '<TYPE>'
      , '<DIMENSION>', '<IDENTIFIER LIST>', '<BIT FIELD LIST>', '<TYPE LIST>'
      , '<CONSTANT VALUE>', '<BIT SPECIFICATION>', '<CONSTANT LIST>'
      , '<EXPRESSION>', '<PROCEDURE HEAD>', '<ENDING>', '<PROCEDURE NAME>'
      , '<STATEMENT>', '<BASIC STATEMENT>', '<IF STATEMENT>', '<ASSIGNMENT>'
      , '<GROUP>', '<RETURN STATEMENT>', '<CALL STATEMENT>'
      , '<GENERATE BLOCK>', '<IF CLAUSE>', '<TRUE PART>', '<EXPRESSION LIST>'
      , '<CASE BODY>', '<GENERATE HEAD>', '<INLINE BODY>', '<LABEL>'
      , '<INLINE STATEMENT>', '<LOGICAL FACTOR>', '<LOGICAL SECONDARY>'
      , '<LOGICAL PRIMARY>', '<STRING EXPRESSION>', '<RELATION>'
      , '<ARITHMETIC EXPRESSION>', '<TERM>', '<ADD>', '<PRIMARY>', '<MULT>'
      , '<CONSTANT>', '<VARIABLE>');

   /* THE DPDA HAS 174 READ STATES. */

   DECLARE READ_START (173) FIXED BIN15 INITIAL (/* 0 */ 0, 2, 14, 16, 28, 39
      , 41, 43, 46, 58, /* 10 */ 60, 62, 64, 66, 68, 70, 81, 87, 91, 94
      , /* 20 */ 96, 101, 110, 113, 116, 125, 127, 138, 141, 143
      , /* 30 */ 150, 152, 154, 156, 167, 169, 171, 173, 175, 177
      , /* 40 */ 187, 196, 199, 208, 217, 226, 237, 246, 248, 257
      , /* 50 */ 259, 261, 269, 275, 278, 282, 288, 297, 299, 301
      , /* 60 */ 303, 307, 309, 311, 313, 316, 319, 321, 325, 328
      , /* 70 */ 337, 339, 342, 345, 347, 349, 352, 361, 370, 372
      , /* 80 */ 374, 377, 379, 381, 384, 387, 389, 401, 404, 413
      , /* 90 */ 416, 425, 434, 442, 444, 446, 448, 454, 460, 464
      , /* 100 */ 467, 476, 485, 494, 497, 506, 508, 510, 514, 516
      , /* 110 */ 518, 527, 532, 534, 536, 538, 547, 552, 554, 556
      , /* 120 */ 565, 567, 569, 571, 580, 591, 594, 605, 607, 609
      , /* 130 */ 617, 621, 624, 627, 630, 633, 642, 646, 649, 652
      , /* 140 */ 656, 660, 662, 664, 666, 669, 679, 688, 690, 692
      , /* 150 */ 704, 713, 725, 728, 730, 733, 735, 737, 741, 750
      , /* 160 */ 752, 754, 756, 758, 761, 763, 765, 774, 783, 794
      , /* 170 */ 798, 801, 804, 816);
```

```
/*  THESE ARE SLR(1) PARSING TABLES  */

   DECLARE NO_TERMINALS LITERALLY '49',
           NO_NTS       LITERALLY '51',
           NO_SYMS      LITERALLY '100';

   DECLARE V(NO_SYMS) CHARACTER(26) INITIAL ('ERROR_SYMBOL', '_|_', ';'
      , 'DECLARE', ',', '<IDENTIFIER>', 'LITERALLY', '<STRING>', '(', ')'
      , 'ENTRY', 'RETURNS', 'FIXED', 'CHARACTER', 'BIT', 'INITIAL', '$'
      , 'PROCEDURE', 'RECURSIVE', 'END PROC', 'EXIT', 'IF', 'THEN', 'ELSE'
      , '=', 'OUTPUT', 'FILE', 'DO', 'END', 'WHILE', 'TO', 'CASE', 'RETURN'
      , 'CALL', 'GENERATE', 'INLINE', 'OR', 'AND', 'NOT', '<', '>', 'NE', '|'
      , '+', '-', '*', '/', 'MOD', '<BUILTIN FUNCTION>', '<NUMBER>'
      , '<PROGRAM>', '<MAIN BODY>', '<DECLARATION LIST>', '<STATEMENT LIST>'
      , '<DECLARATION>', '<DECLARATION STATEMENT>', '<PROCEDURE DEFINITION>'
      , '<DECLARATION ELEMENT LIST>', '<DECLARATION ELEMENT>'
      , '<IDENTIFIER SPECIFICATION>', '<INITIAL LIST>', '<TYPE>'
      , '<DIMENSION>', '<IDENTIFIER LIST>', '<BIT FIELD LIST>', '<TYPE LIST>'
      , '<CONSTANT VALUE>', '<BIT SPECIFICATION>', '<CONSTANT LIST>'
      , '<EXPRESSION>', '<PROCEDURE HEAD>', '<ENDING>', '<PROCEDURE NAME>'
      , '<STATEMENT>', '<BASIC STATEMENT>', '<IF STATEMENT>', '<ASSIGNMENT>'
      , '<GROUP>', '<RETURN STATEMENT>', '<CALL STATEMENT>'
      , '<GENERATE BLOCK>', '<IF CLAUSE>', '<TRUE PART>', '<EXPRESSION LIST>'
      , '<CASE BODY>', '<GENERATE HEAD>', '<INLINE BODY>', '<LABEL>'
      , '<INLINE STATEMENT>', '<LOGICAL FACTOR>', '<LOGICAL SECONDARY>'
      , '<LOGICAL PRIMARY>', '<STRING EXPRESSION>', '<RELATION>'
      , '<ARITHMETIC EXPRESSION>', '<TERM>', '<ADD>', '<PRIMARY>', '<MULT>'
      , '<CONSTANT>', '<VARIABLE>');

   /* THE DPDA HAS 174 READ STATES. */

   DECLARE READ_START (173) FIXED BIN15 INITIAL (/* 0 */ 0, 2, 14, 16, 28, 39
      , 41, 43, 46, 58, /* 10 */ 60, 62, 64, 66, 68, 70, 81, 87, 91, 94
      , /* 20 */ 96, 101, 110, 113, 116, 125, 127, 138, 141, 143
      , /* 30 */ 150, 152, 154, 156, 167, 169, 171, 173, 175, 177
      , /* 40 */ 187, 196, 199, 208, 217, 226, 237, 246, 248, 257
      , /* 50 */ 259, 261, 269, 275, 278, 282, 288, 297, 299, 301
      , /* 60 */ 303, 307, 309, 311, 313, 316, 319, 321, 325, 328
      , /* 70 */ 337, 339, 342, 345, 347, 349, 352, 361, 370, 372
      , /* 80 */ 374, 377, 379, 381, 384, 387, 389, 401, 404, 413
      , /* 90 */ 416, 425, 434, 442, 444, 446, 448, 454, 460, 464
      , /* 100 */ 467, 476, 485, 494, 497, 506, 508, 510, 514, 516
      , /* 110 */ 518, 527, 532, 534, 536, 538, 547, 552, 554, 556
      , /* 120 */ 565, 567, 569, 571, 580, 591, 594, 605, 607, 609
      , /* 130 */ 617, 621, 624, 627, 630, 633, 642, 646, 649, 652
      , /* 140 */ 656, 660, 662, 664, 666, 669, 679, 688, 690, 692
      , /* 150 */ 704, 713, 725, 728, 730, 733, 735, 737, 741, 750
      , /* 160 */ 752, 754, 756, 758, 761, 763, 765, 774, 783, 794
      , /* 170 */ 798, 801, 804, 816);
```

```
DECLARE RD_NO (173) FIXED BIN15 INITIAL (/* 0 */ 1, 11, 1, 11, 10, 1, 1, 2
    , 11, 1, /* 10 */ 1, 1, 1, 1, 1, 10, 5, 3, 2, 1, /* 20 */ 4, 8, 2, 2, 8
    , 1, 10, 2, 1, 6, /* 30 */ 1, 1, 1, 10, 1, 1, 1, 1, 1, 9, /* 40 */ 8, 2
    , 8, 8, 8, 10, 8, 1, 8, 1, /* 50 */ 1, 7, 5, 2, 3, 5, 8, 1, 1, 1
    , /* 60 */ 3, 1, 1, 1, 2, 2, 1, 3, 2, 8, /* 70 */ 1, 2, 2, 1, 1, 2, 8
    , 8, 1, 1, /* 80 */ 2, 1, 1, 2, 2, 1, 11, 2, 8, 2, /* 90 */ 8, 8, 7, 1
    , 1, 1, 5, 5, 3, 2, /* 100 */ 8, 8, 8, 2, 8, 1, 1, 3, 1, 1, /* 110 */ 8
    , 4, 1, 1, 1, 8, 4, 1, 1, 8, /* 120 */ 1, 1, 1, 8, 10, 2, 10, 1, 1, 7
    , /* 130 */ 3, 2, 2, 2, 2, 8, 3, 2, 2, 3, /* 140 */ 3, 1, 1, 1, 2, 9, 8
    , 1, 1, 11, /* 150 */ 8, 11, 2, 1, 2, 1, 1, 3, 8, 1, /* 160 */ 1, 1, 1
    , 2, 1, 1, 8, 8, 10, 3, /* 170 */ 2, 2, 11, 1);

DECLARE SYM_LIST (817) FIXED BIN15 INITIAL (/* 0 */ 1, 0, 2, 3, 5, 20, 21
    , 25, 26, 27, /* 10 */ 32, 33, 34, 0, 1, 0, 2, 3, 5, 20, /* 20 */ 21
    , 25, 26, 27, 32, 33, 34, 0, 2, 5, /* 30 */ 20, 21, 25, 26, 27, 32, 33
    , 34, 0, 2, /* 40 */ 0, 2, 0, 5, 8, 0, 2, 3, 5, 20, /* 50 */ 21, 25, 26
    , 27, 32, 33, 34, 0, 2, 0, /* 60 */ 2, 0, 2, 0, 2, 0, 2, 0, 2, 0
    , /* 70 */ 2, 5, 20, 21, 25, 26, 27, 32, 33, 34, /* 80 */ 0, 2, 8, 12
    , 13, 14, 0, 8, 16, 24, /* 90 */ 0, 8, 24, 0, 8, 0, 2, 5, 29, 31
    , /* 100 */ 0, 5, 7, 8, 38, 43, 44, 48, 49, 0, /* 110 */ 5, 20, 0, 5
    , 35, 0, 5, 7, 8, 38, /* 120 */ 43, 44, 48, 49, 0, 2, 0, 2, 5, 20
    , /* 130 */ 21, 25, 26, 27, 32, 33, 34, 0, 8, 24, /* 140 */ 0, 4, 0, 6
    , 8, 10, 12, 13, 14, 0, /* 150 */ 15, 0, 5, 0, 19, 0, 2, 5, 20, 21
    , /* 160 */ 25, 26, 27, 32, 33, 34, 0, 23, 0, 2, /* 170 */ 0, 5, 0, 8
    , 0, 8, 0, 5, 7, 8, /* 180 */ 26, 38, 43, 44, 48, 49, 0, 5, 7, 8
    , /* 190 */ 38, 43, 44, 48, 49, 0, 17, 18, 0, 5, /* 200 */ 7, 8, 38, 43
    , 44, 48, 49, 0, 5, 7, /* 210 */ 8, 38, 43, 44, 48, 49, 0, 5, 7, 8
    , /* 220 */ 38, 43, 44, 48, 49, 0, 2, 5, 20, 21, /* 230 */ 25, 26, 27
    , 32, 33, 34, 0, 5, 7, 8, /* 240 */ 38, 43, 44, 48, 49, 0, 24, 0, 5, 7
    , /* 250 */ 8, 38, 43, 44, 48, 49, 0, 36, 0, 37, /* 260 */ 0, 5, 7, 8
    , 43, 44, 48, 49, 0, 24, /* 270 */ 39, 40, 41, 42, 0, 43, 44, 0, 45, 46
    , /* 280 */ 47, 0, 5, 7, 8, 48, 49, 0, 5, 7, /* 290 */ 8, 38, 43, 44
    , 48, 49, 0, 8, 0, 8, /* 300 */ 0, 8, 0, 5, 28, 35, 0, 8, 0, 35
    , /* 310 */ 0, 16, 0, 22, 36, 0, 5, 8, 0, 7, /* 320 */ 0, 12, 13, 14, 0
    , 8, 11, 0, 5, 7, /* 330 */ 8, 38, 43, 44, 48, 49, 0, 8, 0, 4
    , /* 340 */ 9, 0, 4, 9, 0, 8, 0, 5, 0, 4, /* 350 */ 9, 0, 5, 7, 8, 38
    , 43, 44, 48, 49, /* 360 */ 0, 5, 7, 8, 38, 43, 44, 48, 49, 0
    , /* 370 */ 36, 0, 8, 0, 9, 36, 0, 17, 0, 36, /* 380 */ 0, 9, 36, 0, 4
    , 9, 0, 36, 0, 2, /* 390 */ 5, 20, 21, 25, 26, 27, 28, 32, 33, 34
    , /* 400 */ 0, 2, 36, 0, 5, 7, 8, 38, 43, 44, /* 410 */ 48, 49, 0, 2
    , 36, 0, 5, 7, 8, 38, /* 420 */ 43, 44, 48, 49, 0, 5, 7, 8, 38, 43
    , /* 430 */ 44, 48, 49, 0, 5, 7, 8, 43, 44, 48, /* 440 */ 49, 0, 42, 0
    , 24, 0, 24, 0, 5, 7, /* 450 */ 8, 48, 49, 0, 5, 7, 8, 48, 49, 0
    , /* 460 */ 45, 46, 47, 0, 9, 36, 0, 5, 7, 8, /* 470 */ 38, 43, 44, 48
    , 49, 0, 5, 7, 8, 38, /* 480 */ 43, 44, 48, 49, 0, 5, 7, 8, 38, 43
    , /* 490 */ 44, 48, 49, 0, 28, 35, 0, 5, 7, 8, /* 500 */ 38, 43, 44, 48
    , 49, 0, 8, 0, 8, 0, /* 510 */ 12, 13, 14, 0, 9, 0, 36, 0, 5, 7
    , /* 520 */ 8, 38, 43, 44, 48, 49, 0, 8, 12, 13, /* 530 */ 14, 0, 5, 0
    , 8, 0, 5, 0, 5, 7, /* 540 */ 8, 38, 43, 44, 48, 49, 0, 2, 12, 13
    , /* 550 */ 14, 0, 9, 0, 9, 0, 5, 7, 8, 38, /* 560 */ 43, 44, 48, 49, 0
    , 24, 0, 24, 0, 24, /* 570 */ 0, 5, 7, 8, 38, 43, 44, 48, 49, 0
, /* 580 */ 2, 5, 20, 21, 25, 26, 27, 32, 33, 34, /* 590 */ 0, 30, 36
, 0, 2, 5, 20, 21, 25, 26, /* 600 */ 27, 32, 33, 34, 0, 37, 0, 42, 0, 5
, /* 610 */ 7, 8, 43, 44, 48, 49, 0, 45, 46, 47, /* 620 */ 0, 4, 9, 0
, 4, 9, 0, 4, 9, 0, /* 630 */ 4, 9, 0, 5, 7, 8, 38, 43, 44, 48
, /* 640 */ 49, 0, 12, 13, 14, 0, 4, 9, 0, 4, /* 650 */ 9, 0, 12, 13
, 14, 0, 12, 13, 14, 0, /* 660 */ 8, 0, 9, 0, 2, 0, 4, 9, 0, 5
, /* 670 */ 7, 8, 26, 38, 43, 44, 48, 49, 0, 5, /* 680 */ 7, 8, 38, 43
, 44, 48, 49, 0, 5, 0, /* 690 */ 36, 0, 2, 5, 20, 21, 25, 26, 27, 28
, /* 700 */ 32, 33, 34, 0, 5, 7, 8, 38, 43, 44, /* 710 */ 48, 49, 0, 2
, 5, 20, 21, 25, 26, 27, /* 720 */ 28, 32, 33, 34, 0, 43, 44, 0, 2, 0
, /* 730 */ 4, 9, 0, 9, 0, 11, 0, 12, 13, 14, /* 740 */ 0, 5, 7, 8, 38
, 43, 44, 48, 49, 0, /* 750 */ 36, 0, 8, 0, 36, 0, 8, 0, 2, 36
, /* 760 */ 0, 2, 0, 8, 0, 5, 7, 8, 38, 43, /* 770 */ 44, 48, 49, 0, 5
, 7, 8, 38, 43, 44, /* 780 */ 48, 49, 0, 2, 5, 20, 21, 25, 26, 27
, /* 790 */ 32, 33, 34, 0, 12, 13, 14, 0, 4, 9, /* 800 */ 0, 9, 36, 0
, 2, 5, 20, 21, 25, 26, /* 810 */ 27, 28, 32, 33, 34, 0, 9, 0);

DECLARE STATE_LIST (817) FIXED BIN15 INITIAL (/* 0 */ 1, 1791, 568, 7, 17
    , 13, 24, 18, 19, 20, /* 10 */ 770, 22, 25, 1791, 637, 1791, 568, 7, 17
    , 13, /* 20 */ 24, 18, 19, 20, 770, 22, 25, 1791, 568, 27, /* 30 */ 13
    , 24, 18, 19, 20, 770, 22, 25, 1791, 519, /* 40 */ 1791, 520, 1791, 29
    , 31, 1791, 568, 7, 17, 13, /* 50 */ 24, 18, 19, 20, 770, 22, 25, 1791
    , 562, 1791, /* 60 */ 563, 1791, 564, 1791, 565, 1791, 566, 1791, 567
    , 1791, /* 70 */ 568, 27, 13, 24, 18, 19, 20, 770, 22, 25
    , /* 80 */ 1791, 550, 36, 534, 37, 38, 1791, 40, 41, 39, /* 90 */ 1791
    , 43, 42, 1791, 44, 1791, 45, 47, 46, 48, /* 100 */ 1791, 780, 635, 56
    , 51, 619, 620, 781, 636, 1791, /* 110 */ 782, 591, 1791, 63, 61, 1791
    , 780, 635, 56, 51, /* 120 */ 619, 620, 781, 636, 1791, 594, 1791, 568
    , 27, 13, /* 130 */ 24, 18, 19, 20, 770, 22, 25, 1791, 40, 39
    , /* 140 */ 1791, 65, 1791, 66, 69, 68, 534, 37, 38, 1791, /* 150 */ 70
    , 1791, 783, 1791, 784, 1791, 568, 27, 13, 24, /* 160 */ 18, 19, 20
    , 770, 22, 25, 1791, 572, 1791, 551, /* 170 */ 1791, 538, 1791, 76
    , 1791, 77, 1791, 780, 635, 56, /* 180 */ 79, 51, 619, 620, 781, 636
    , 1791, 780, 635, 56, /* 190 */ 51, 619, 620, 781, 636, 1791, 554, 81
    , 1791, 780, /* 200 */ 635, 56, 51, 619, 620, 781, 636, 1791, 780, 635
    , /* 210 */ 56, 51, 619, 620, 781, 636, 1791, 780, 635, 56
    , /* 220 */ 51, 619, 620, 781, 636, 1791, 568, 27, 13, 24, /* 230 */ 18
    , 19, 20, 770, 22, 25, 1791, 780, 635, 56, /* 240 */ 51, 619, 620, 781
    , 636, 1791, 88, 1791, 780, 635, /* 250 */ 56, 51, 619, 620, 781, 636
    , 1791, 90, 1791, 91, /* 260 */ 1791, 780, 635, 56, 619, 620, 781, 636
    , 1791, 608, /* 270 */ 788, 789, 613, 93, 1791, 619, 620, 1791, 623
    , 624, /* 280 */ 625, 1791, 780, 635, 56, 781, 636, 1791, 780, 635
    , /* 290 */ 56, 51, 619, 620, 781, 636, 1791, 100, 1791, 101
    , /* 300 */ 1791, 102, 1791, 63, 592, 61, 1791, 104, 1791, 105
    , /* 310 */ 1791, 599, 1791, 571, 90, 1791, 29, 31, 1791, 524
    , /* 320 */ 1791, 534, 37, 38, 1791, 107, 106, 1791, 780, 635
    , /* 330 */ 56, 51, 619, 620, 781, 636, 1791, 110, 1791, 112
    , /* 340 */ 111, 1791, 114, 113, 1791, 115, 1791, 557, 1791, 112
    , /* 350 */ 116, 1791, 780, 635, 56, 51, 619, 620, 781, 636
    , /* 360 */ 1791, 780, 635, 56, 51, 619, 620, 781, 636, 1791
    , /* 370 */ 90, 1791, 119, 1791, 120, 90, 1791, 555, 1791, 90
    , /* 380 */ 1791, 121, 90, 1791, 123, 122, 1791, 90, 1791, 568
, /* 390 */ 27, 13, 24, 18, 19, 20, 581, 770, 22, 25, /* 400 */ 1791
, 124, 90, 1791, 780, 635, 56, 51, 619, 620, /* 410 */ 781, 636, 1791
, 126, 90, 1791, 780, 635, 56, 51, /* 420 */ 619, 620, 781, 636, 1791
, 780, 635, 56, 51, 619, /* 430 */ 620, 781, 636, 1791, 780, 635, 56
, 619, 620, 781, /* 440 */ 636, 1791, 129, 1791, 611, 1791, 612, 1791
, 780, 635, /* 450 */ 56, 781, 636, 1791, 780, 635, 56, 781, 636, 1791
, /* 460 */ 623, 624, 625, 1791, 628, 90, 1791, 780, 635, 56
, /* 470 */ 51, 619, 620, 781, 636, 1791, 780, 635, 56, 51
, /* 480 */ 619, 620, 781, 636, 1791, 780, 635, 56, 51, 619
, /* 490 */ 620, 781, 636, 1791, 593, 105, 1791, 780, 635, 56
, /* 500 */ 51, 619, 620, 781, 636, 1791, 135, 1791, 136, 1791
, /* 510 */ 534, 37, 38, 1791, 537, 1791, 90, 1791, 780, 635
, /* 520 */ 56, 51, 619, 620, 781, 636, 1791, 69, 534, 37, /* 530 */ 38
, 1791, 539, 1791, 69, 1791, 141, 1791, 780, 635, /* 540 */ 56, 51, 619
, 620, 781, 636, 1791, 552, 534, 37, /* 550 */ 38, 1791, 535, 1791, 536
, 1791, 780, 635, 56, 51, /* 560 */ 619, 620, 781, 636, 1791, 145, 1791
, 146, 1791, 147, /* 570 */ 1791, 780, 635, 56, 51, 619, 620, 781, 636
, 1791, /* 580 */ 568, 27, 13, 24, 18, 19, 20, 770, 22, 25
, /* 590 */ 1791, 150, 90, 1791, 568, 27, 13, 24, 18, 19, /* 600 */ 20
, 770, 22, 25, 1791, 91, 1791, 93, 1791, 780, /* 610 */ 635, 56, 619
, 620, 781, 636, 1791, 623, 624, 625, /* 620 */ 1791, 123, 630, 1791
, 123, 632, 1791, 123, 590, 1791, /* 630 */ 123, 153, 1791, 780, 635
, 56, 51, 619, 620, 781, /* 640 */ 636, 1791, 534, 37, 38, 1791, 157
, 156, 1791, 158, /* 650 */ 545, 1791, 534, 37, 38, 1791, 534, 37, 38
, 1791, /* 660 */ 115, 1791, 542, 1791, 553, 1791, 123, 579, 1791, 780
, /* 670 */ 635, 56, 160, 51, 619, 620, 781, 636, 1791, 780
, /* 680 */ 635, 56, 51, 619, 620, 781, 636, 1791, 799, 1791
, /* 690 */ 90, 1791, 568, 27, 13, 24, 18, 19, 20, 582, /* 700 */ 770
, 22, 25, 1791, 780, 635, 56, 51, 619, 620, /* 710 */ 781, 636, 1791
, 568, 27, 13, 24, 18, 19, 20, /* 720 */ 584, 770, 22, 25, 1791, 619
, 620, 1791, 597, 1791, /* 730 */ 123, 164, 1791, 532, 1791, 165, 1791
, 534, 37, 38, /* 740 */ 1791, 780, 635, 56, 51, 619, 620, 781, 636
, 1791, /* 750 */ 90, 1791, 166, 1791, 90, 1791, 167, 1791, 168, 90
, /* 760 */ 1791, 598, 1791, 169, 1791, 780, 635, 56, 51, 619
, /* 770 */ 620, 781, 636, 1791, 780, 635, 56, 51, 619, 620
, /* 780 */ 781, 636, 1791, 568, 27, 13, 24, 18, 19, 20, /* 790 */ 770
, 22, 25, 1791, 534, 37, 38, 1791, 123, 580, /* 800 */ 1791, 578, 90
, 1791, 568, 27, 13, 24, 18, 19, /* 810 */ 20, 583, 770, 22, 25, 1791
, 533, 1791);
, /* 390 */ 27, 13, 24, 18, 19, 20, 581, 770, 22, 25, /* 400 */ 1791
, 124, 90, 1791, 780, 635, 56, 51, 619, 620, /* 410 */ 781, 636, 1791
, 126, 90, 1791, 780, 635, 56, 51, /* 420 */ 619, 620, 781, 636, 1791
, 780, 635, 56, 51, 619, /* 430 */ 620, 781, 636, 1791, 780, 635, 56
, 619, 620, 781, /* 440 */ 636, 1791, 129, 1791, 611, 1791, 612, 1791
, 780, 635, /* 450 */ 56, 781, 636, 1791, 780, 635, 56, 781, 636, 1791
, /* 460 */ 623, 624, 625, 1791, 628, 90, 1791, 780, 635, 56
, /* 470 */ 51, 619, 620, 781, 636, 1791, 780, 635, 56, 51
, /* 480 */ 619, 620, 781, 636, 1791, 780, 635, 56, 51, 619
, /* 490 */ 620, 781, 636, 1791, 593, 105, 1791, 780, 635, 56
, /* 500 */ 51, 619, 620, 781, 636, 1791, 135, 1791, 136, 1791
, /* 510 */ 534, 37, 38, 1791, 537, 1791, 90, 1791, 780, 635
, /* 520 */ 56, 51, 619, 620, 781, 636, 1791, 69, 534, 37, /* 530 */ 38
, 1791, 539, 1791, 69, 1791, 141, 1791, 780, 635, /* 540 */ 56, 51, 619
, 620, 781, 636, 1791, 552, 534, 37, /* 550 */ 38, 1791, 535, 1791, 536
, 1791, 780, 635, 56, 51, /* 560 */ 619, 620, 781, 636, 1791, 145, 1791
, 146, 1791, 147, /* 570 */ 1791, 780, 635, 56, 51, 619, 620, 781, 636
, 1791, /* 580 */ 568, 27, 13, 24, 18, 19, 20, 770, 22, 25
, /* 590 */ 1791, 150, 90, 1791, 568, 27, 13, 24, 18, 19, /* 600 */ 20
, 770, 22, 25, 1791, 91, 1791, 93, 1791, 780, /* 610 */ 635, 56, 619
, 620, 781, 636, 1791, 623, 624, 625, /* 620 */ 1791, 123, 630, 1791
, 123, 632, 1791, 123, 590, 1791, /* 630 */ 123, 153, 1791, 780, 635
, 56, 51, 619, 620, 781, /* 640 */ 636, 1791, 534, 37, 38, 1791, 157
, 156, 1791, 158, /* 650 */ 545, 1791, 534, 37, 38, 1791, 534, 37, 38
, 1791, /* 660 */ 115, 1791, 542, 1791, 553, 1791, 123, 579, 1791, 780
, /* 670 */ 635, 56, 160, 51, 619, 620, 781, 636, 1791, 780
, /* 680 */ 635, 56, 51, 619, 620, 781, 636, 1791, 799, 1791
, /* 690 */ 90, 1791, 568, 27, 13, 24, 18, 19, 20, 582, /* 700 */ 770
, 22, 25, 1791, 780, 635, 56, 51, 619, 620, /* 710 */ 781, 636, 1791
, 568, 27, 13, 24, 18, 19, 20, /* 720 */ 584, 770, 22, 25, 1791, 619
, 620, 1791, 597, 1791, /* 730 */ 123, 164, 1791, 532, 1791, 165, 1791
, 534, 37, 38, /* 740 */ 1791, 780, 635, 56, 51, 619, 620, 781, 636
, 1791, /* 750 */ 90, 1791, 166, 1791, 90, 1791, 167, 1791, 168, 90
, /* 760 */ 1791, 598, 1791, 169, 1791, 780, 635, 56, 51, 619
, /* 770 */ 620, 781, 636, 1791, 780, 635, 56, 51, 619, 620
, /* 780 */ 781, 636, 1791, 568, 27, 13, 24, 18, 19, 20, /* 790 */ 770
, 22, 25, 1791, 534, 37, 38, 1791, 123, 580, /* 800 */ 1791, 578, 90
, 1791, 568, 27, 13, 24, 18, 19, /* 810 */ 20, 583, 770, 22, 25, 1791
, 533, 1791);
```

```
/* THE DPDA HAS 125 REDUCE STATES. */

DECLARE NO_TO_POP (125) FIXED BIN15 INITIAL (/* 0 */ 0, 0, 1, 0, 0, 0, 1
    , 1, 1, 1, /* 10 */ 0, 2, 2, 0, 1, 1, 2, 3, 4, 4, /* 20 */ 5, 8, 0, 3
    , 3, 2, 0, 2, 0, 2, /* 30 */ 3, 0, 2, 3, 0, 2, 0, 2, 1, 2, /* 40 */ 4
    , 5, 2, 3, 0, 1, 0, 1, 0, 0, /* 50 */ 1, 1, 1, 1, 1, 1, 0, 1, 2, 2
    , /* 60 */ 1, 2, 5, 2, 5, 5, 8, 5, 8, 3, /* 70 */ 5, 8, 5, 0, 1, 0, 1
    , 1, 4, 1, /* 80 */ 2, 3, 1, 0, 1, 4, 5, 1, 0, 2, /* 90 */ 0, 2, 0, 1
    , 0, 2, 0, 0, 0, 1, /* 100 */ 1, 0, 0, 3, 0, 2, 1, 0, 0, 0, /* 110 */ 2
    , 0, 0, 0, 0, 0, 2, 0, 3, 0, /* 120 */ 3, 0, 2, 0, 0, 2);

DECLARE REDUCE_SUCC (125) FIXED BIN15 INITIAL (/* 0 */ 0, 2, 1280, 1280
    , 1280, 768, 768, 1282, 1282, 5, /* 10 */ 772, 772, 1283, 1283, 1283
    , 773, 773, 773, 773, 773, /* 20 */ 773, 773, 1284, 1284, 1284, 1285
    , 1286, 1286, 72, 72, /* 30 */ 1288, 137, 137, 526, 138, 138, 1287, 6
    , 8, 8, /* 40 */ 8, 8, 16, 16, 549, 549, 1281, 1281, 1290, 1290
    , /* 50 */ 1291, 1291, 1291, 1291, 1291, 1291, 1291, 561, 561, 15
    , /* 60 */ 33, 9, 9, 9, 9, 9, 9, 9, 9, 10, /* 70 */ 10, 10, 10, 151
    , 151, 11, 11, 12, 12, 12, /* 80 */ 14, 14, 23, 60, 60, 1294, 1294
    , 1293, 1289, 1289, /* 90 */ 1295, 1295, 1296, 1296, 1297, 1297, 92, 92
    , 92, 92, /* 100 */ 92, 92, 1298, 1298, 1299, 1299, 1299, 1301, 1301
    , 1300, /* 110 */ 1300, 97, 97, 97, 1302, 1302, 1302, 627, 627, 627
    , /* 120 */ 627, 1292, 1292, 626, 626, 1792);

/* THE DPDA HAS 32 LOOK-AHEAD STATES. */

DECLARE LA_SYM_NO (31) FIXED BIN15 INITIAL (/* 0 */ 0, 0, 0, 0, 0, 0, 0, 0
    , 0, 0, /* 10 */ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 20 */ 0, 0, 0, 0, 0
    , 0, 0, 0, 0, 0, /* 30 */ 0, 0);

DECLARE SUCC_STATE (31) FIXED BIN15 INITIAL (/* 0 */ 515, 516, 587, 514
    , 521, 525, 560, 588, 600, 606, /* 10 */ 614, 616, 629, 631, 589, 538
    , 556, 573, 575, 633, /* 20 */ 609, 610, 618, 548, 601, 607, 617, 634
    , 615, 574, /* 30 */ 576, 577);

DECLARE FAIL_STATE (31) FIXED BIN15 INITIAL (/* 0 */ 3, 4, 21, 26, 28, 30
    , 34, 49, 50, 52, /* 10 */ 53, 54, 57, 58, 59, 73, 74, 78, 82, 85
    , /* 20 */ 94, 95, 98, 109, 127, 128, 130, 148, 152, 159, /* 30 */ 161
    , 162);


DECLARE LA_TABLE (31) FIXED BIN15 INITIAL (
    ^(1)010000000000000000100000000000000000000000000000000000000^
    /* <MAIN BODY> */,
    ^(1)010000000000000000100000000000000000000000000000000000000^
    /* <MAIN BODY> */,
    ^(1)001000000000000000000000000000000000000000000000000000000^
    /* <RETURN STATEMENT> */,
    ^(1)010000000000000000100000000000000000000000000000000000000^
    /* <MAIN BODY> */,
    ^(1)001000000000000000000000000000000000000000000000000000000^
    /* <DECLARATION STATEMENT> */,
    ^(1)001010000000000000000000000000000000000000000000000000000^
```

```
/* THE DPDA HAS 125 REDUCE STATES. */

DECLARE NO_TO_POP (125) FIXED BIN15 INITIAL (/* 0 */ 0, 0, 1, 0, 0, 0, 1
   , 1, 1, 1, /* 10 */ 0, 2, 2, 0, 1, 1, 2, 3, 4, 4, /* 20 */ 5, 8, 0, 3
   , 3, 2, 0, 2, 0, 2, /* 30 */ 3, 0, 2, 3, 0, 2, 0, 2, 1, 2, /* 40 */ 4
   , 5, 2, 3, 0, 1, 0, 1, 0, 0, /* 50 */ 1, 1, 1, 1, 1, 1, 0, 1, 2, 2
   , /* 60 */ 1, 2, 5, 2, 5, 5, 8, 5, 8, 3, /* 70 */ 5, 8, 5, 0, 1, 0, 1
   , 1, 4, 1, /* 80 */ 2, 3, 1, 0, 1, 4, 5, 1, 0, 2, /* 90 */ 0, 2, 0, 1
   , 0, 2, 0, 0, 0, 1, /* 100 */ 1, 0, 0, 3, 0, 2, 1, 0, 0, 0, /* 110 */ 2
   , 0, 0, 0, 0, 0, 2, 0, 3, 0, /* 120 */ 3, 0, 2, 0, 0, 2);

DECLARE REDUCE_SUCC (125) FIXED BIN15 INITIAL (/* 0 */ 0, 2, 1280, 1280
   , 1280, 768, 768, 1282, 1282, 5, /* 10 */ 772, 772, 1283, 1283, 1283
   , 773, 773, 773, 773, 773, /* 20 */ 773, 773, 1284, 1284, 1284, 1285
   , 1286, 1286, 72, 72, /* 30 */ 1288, 137, 137, 526, 138, 138, 1287, 6
   , 8, 8, /* 40 */ 8, 8, 16, 16, 549, 549, 1281, 1281, 1290, 1290
   , /* 50 */ 1291, 1291, 1291, 1291, 1291, 1291, 1291, 561, 561, 15
   , /* 60 */ 33, 9, 9, 9, 9, 9, 9, 9, 9, 10, /* 70 */ 10, 10, 10, 151
   , 151, 11, 11, 12, 12, 12, /* 80 */ 14, 14, 23, 60, 60, 1294, 1294
   , 1293, 1289, 1289, /* 90 */ 1295, 1295, 1296, 1296, 1297, 1297, 92, 92
   , 92, 92, /* 100 */ 92, 92, 1298, 1298, 1299, 1299, 1299, 1301, 1301
   , 1300, /* 110 */ 1300, 97, 97, 97, 1302, 1302, 1302, 627, 627, 627
   , /* 120 */ 627, 1292, 1292, 626, 626, 1792);

/* THE DPDA HAS 32 LOOK-AHEAD STATES. */

DECLARE LA_SYM_NO (31) FIXED BIN15 INITIAL (/* 0 */ 0, 0, 0, 0, 0, 0, 0, 0
   , 0, 0, /* 10 */ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 20 */ 0, 0, 0, 0, 0
   , 0, 0, 0, 0, 0, /* 30 */ 0, 0);

DECLARE SUCC_STATE (31) FIXED BIN15 INITIAL (/* 0 */ 515, 516, 587, 514
   , 521, 525, 560, 588, 600, 606, /* 10 */ 614, 616, 629, 631, 589, 538
   , 556, 573, 575, 633, /* 20 */ 609, 610, 618, 548, 601, 607, 617, 634
   , 615, 574, /* 30 */ 576, 577);

DECLARE FAIL_STATE (31) FIXED BIN15 INITIAL (/* 0 */ 3, 4, 21, 26, 28, 30
   , 34, 49, 50, 52, /* 10 */ 53, 54, 57, 58, 59, 73, 74, 78, 82, 85
   , /* 20 */ 94, 95, 98, 109, 127, 128, 130, 148, 152, 159, /* 30 */ 161
   , 162);


DECLARE LA_TABLE (31) FIXED BIN15 INITIAL (
    ^(1)0100000000000000000010000000000000000000000000000000000000000000^
    /* <MAIN BODY> */,
    ^(1)0100000000000000000010000000000000000000000000000000000000000000^
    /* <MAIN BODY> */,
    ^(1)0010000000000000000000000000000000000000000000000000000000000000^
    /* <RETURN STATEMENT> */,
    ^(1)0100000000000000000010000000000000000000000000000000000000000000^
    /* <MAIN BODY> */,
    ^(1)0010000000000000000000000000000000000000000000000000000000000000^
    /* <DECLARATION STATEMENT> */,
    ^(1)0010100000000000000000000000000000000000000000000000000000000000^
/* <DECLARATION ELEMENT> */,
^(1)0110010000000000000011100011110001110000000000000000000000000000^
/* <STATEMENT> */,
^(1)0010000000000000000000000000000000000000000000000000000000000000^
/* <RETURN STATEMENT> */,
^(1)0010100001000000000000100000001000001000000000000000000000000000^
/* <EXPRESSION> */,
^(1)0010100001000000000000100000001000001100000000000000000000000000^
/* <LOGICAL PRIMARY> */,
^(1)0010100001000000000000101000000100000110111110000000000000000000^
/* <STRING EXPRESSION> */,
^(1)0010100001000000000000101000000100000110111111000000000000000000^
/* <ARITHMETIC EXPRESSION> */,
^(1)0010100001000000000000101000000100000110111111111000000000000000^
/* <VARIABLE> */,
^(1)0010100001000000000000101000000100000110111111111000000000000000^
/* <VARIABLE> */,
^(1)0010000000000000000000000000000000000000000000000000000000000000^
/* <CALL STATEMENT> */,
^(1)0000100001000000000000000000000000000000000000000000000000000000^
/* <IDENTIFIER LIST> */,
^(1)0010000000000000000000000000000000000000000000000000000000000000^
/* <ENDING> */,
^(1)0010000000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */,
^(1)0010000000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */,
^(1)0000100001000000000000000000000000000000000000000000000000000000^
/* <EXPRESSION LIST> */,
^(1)0000010110000000000000000000000000000000001100011000000000000000^
/* <RELATION> */,
^(1)0000010110000000000000000000000000000000001100011000000000000000^
/* <RELATION> */,
^(1)0010100001000000000000101000000100000110111111000000000000000000^
/* <ARITHMETIC EXPRESSION> */,
^(1)0000100001000000000000000000000000000000000000000000000000000000^
/* <CONSTANT VALUE> */,
^(1)0010100001000000000000100000001000001000000000000000000000000000^
/* <EXPRESSION> */,
^(1)0010100001000000000000100000001000001100000000000000000000000000^
/* <LOGICAL PRIMARY> */,
^(1)0010100001000000000000101000000100000110111111000000000000000000^
/* <ARITHMETIC EXPRESSION> */,
^(1)0000100001000000000000000000000000000000000000000000000000000000^
/* <EXPRESSION LIST> */,
^(1)0010100001000000000000101000000100000110111100000000000000000000^
/* <STRING EXPRESSION> */,
^(1)0010000000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */,
^(1)0010000000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */,
^(1)0010000000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */);
/* <DECLARATION ELEMENT> */,
^(1)011001000000000000001110001111000111000000000000000000000000^
/* <STATEMENT> */,
^(1)001000000000000000000000000000000000000000000000000000000000^
/* <RETURN STATEMENT> */,
^(1)001010000100000000000100000001000001000000000000000000000000^
/* <EXPRESSION> */,
^(1)001010000100000000000100000001000001100000000000000000000000^
/* <LOGICAL PRIMARY> */,
^(1)001010000100000000000101000001000001101111100000000000000000^
/* <STRING EXPRESSION> */,
^(1)001010000100000000000101000001000001101111110000000000000000^
/* <ARITHMETIC EXPRESSION> */,
^(1)001010000100000000000101000001000001101111111110000000000000^
/* <VARIABLE> */,
^(1)001010000100000000000101000001000001101111111110000000000000^
/* <VARIABLE> */,
^(1)001000000000000000000000000000000000000000000000000000000000^
/* <CALL STATEMENT> */,
^(1)000010000100000000000000000000000000000000000000000000000000^
/* <IDENTIFIER LIST> */,
^(1)001000000000000000000000000000000000000000000000000000000000^
/* <ENDING> */,
^(1)001000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */,
^(1)001000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */,
^(1)000010000100000000000000000000000000000000000000000000000000^
/* <EXPRESSION LIST> */,
^(1)000001011000000000000000000000000000000011000110000000000000^
/* <RELATION> */,
^(1)000001011000000000000000000000000000000011000110000000000000^
/* <RELATION> */,
^(1)001010000100000000000101000001000001101111110000000000000000^
/* <ARITHMETIC EXPRESSION> */,
^(1)000010000100000000000000000000000000000000000000000000000000^
/* <CONSTANT VALUE> */,
^(1)001010000100000000000100000001000001000000000000000000000000^
/* <EXPRESSION> */,
^(1)001010000100000000000100000001000001100000000000000000000000^
/* <LOGICAL PRIMARY> */,
^(1)001010000100000000000101000001000001101111110000000000000000^
/* <ARITHMETIC EXPRESSION> */,
^(1)000010000100000000000000000000000000000000000000000000000000^
/* <EXPRESSION LIST> */,
^(1)001010000100000000000101000001000001101111000000000000000000^
/* <STRING EXPRESSION> */,
^(1)001000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */,
^(1)001000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */,
^(1)001000000000000000000000000000000000000000000000000000000000^
/* <ASSIGNMENT> */);
/* THE DPDA HAS 23LOOK-BACK STATES. */

DECLARE LB_START (22) FIXED BIN15 INITIAL (/* 0 */ 0, 2, 7, 9, 11, 22, 25
    , 27, 33, 35, /* 10 */ 57, 67, 69, 77, 79, 81, 83, 85, 87, 89
    , /* 20 */ 91, 94, 97);

DECLARE LB_NO (22) FIXED BIN15 INITIAL (/* 0 */ 1, 4, 1, 1, 10, 2, 1, 5, 1
    , 21, /* 10 */ 9, 1, 7, 1, 1, 1, 1, 1, 1, 1, /* 20 */ 2, 2, 1);

DECLARE LB_STATE (98) FIXED BIN15 INITIAL (/* 0 */ 1, 0, 3, 45, 124, 168
    , 0, 3, 0, 7, /* 10 */ 0, 16, 29, 67, 107, 111, 116, 136, 139, 140
    , /* 20 */ 157, 0, 29, 111, 0, 31, 0, 69, 76, 77, /* 30 */ 110, 115, 0
    , 31, 0, 21, 24, 39, 40, 42, /* 40 */ 43, 46, 48, 56, 88, 123, 145, 146
    , 150, 167, /* 50 */ 69, 76, 77, 110, 115, 158, 0, 15, 33, 126
    , /* 60 */ 151, 4, 26, 86, 149, 172, 0, 15, 0, 44, /* 70 */ 100, 101
    , 102, 104, 119, 135, 0, 23, 0, 23, /* 80 */ 0, 90, 0, 91, 0, 51, 0, 92
    , 0, 129, /* 90 */ 0, 55, 96, 0, 53, 152, 0, 97, 0);

DECLARE RESUME_STATE (98) FIXED BIN15 INITIAL (/* 0 */ 513, 32, 771, 86
    , 149, 172, 769, 518, 517, 522, /* 10 */ 523, 35, 527, 528, 543, 529
    , 143, 155, 530, 531, /* 20 */ 544, 173, 67, 139, 140, 71, 75, 108, 117
    , 118, /* 30 */ 546, 142, 547, 540, 541, 775, 64, 785, 80, 786
    , /* 40 */ 83, 87, 89, 99, 125, 795, 797, 798, 163, 171, /* 50 */ 791
    , 791, 791, 791, 791, 791, 787, 569, 570, 585, /* 60 */ 586, 559, 559
    , 559, 559, 559, 558, 774, 560, 84, /* 70 */ 131, 132, 133, 134, 144
    , 154, 170, 62, 103, 595, /* 80 */ 596, 792, 776, 603, 602, 605, 604
    , 793, 777, 796, /* 90 */ 778, 790, 794, 779, 96, 96, 55, 622, 621);

/* THE SYMBOL ACCESSING THE STATES. */

DECLARE SYM_BEFORE_READ (173) FIXED BIN15 INITIAL (/* 0 */ 0, 1, 50, 52
    , 53, 55, 56, 3, 70, 76, /* 10 */ 77, 78, 79, 20, 80, 81, 72, 5, 25, 26
    , /* 20 */ 27, 32, 33, 85, 21, 34, 53, 5, 57, 5, /* 30 */ 59, 8, 51, 82
    , 74, 61, 8, 13, 14, 24, /* 40 */ 8, 16, 24, 8, 8, 2, 29, 5, 31, 69
    , /* 50 */ 89, 38, 92, 94, 95, 96, 8, 5, 48, 5, /* 60 */ 86, 35, 87, 5
    , 69, 4, 6, 62, 10, 8, /* 70 */ 15, 63, 64, 5, 19, 63, 8, 8, 69, 26
    , /* 80 */ 69, 18, 69, 69, 83, 69, 53, 69, 24, 69, /* 90 */ 36, 37, 93
    , 42, 39, 40, 96, 98, 95, 69, /* 100 */ 8, 8, 8, 87, 8, 35, 11, 8, 66
    , 69, /* 110 */ 8, 9, 4, 9, 4, 8, 9, 66, 66, 8, /* 120 */ 9, 9, 9, 4, 2
    , 69, 2, 89, 92, 42, /* 130 */ 95, 83, 83, 83, 83, 8, 8, 65, 68, 62
    , /* 140 */ 62, 5, 66, 61, 83, 24, 24, 24, 69, 53, /* 150 */ 30, 84, 94
    , 9, 83, 61, 9, 4, 4, 69, /* 160 */ 26, 69, 5, 69, 9, 11, 8, 8, 2, 8
    , /* 170 */ 83, 69, 53, 61);
DECLARE SYM_BEFORE_LA (31) FIXED BIN15 INITIAL (/* 0 */ 52, 53
    , /* 770 */ 32, 53, 57, 59, 74, 69, 89, 92, 94, 95, /* 780 */ 5, 48, 5
    , 5, 19, 69, 69, 69, 39, 40, /* 790 */ 95, 69, 89, 92, 95, 69, 94, 69
    , 69, 5);

/* END OF SLR(1) PARSING TABLES . */
/* THE DPDA HAS 23LOOK-BACK STATES. */

DECLARE LB_START (22) FIXED BIN15 INITIAL (/* 0 */ 0, 2, 7, 9, 11, 22, 25
   , 27, 33, 35, /* 10 */ 57, 67, 69, 77, 79, 81, 83, 85, 87, 89
   , /* 20 */ 91, 94, 97);

DECLARE LB_NO (22) FIXED BIN15 INITIAL (/* 0 */ 1, 4, 1, 1, 10, 2, 1, 5, 1
   , 21, /* 10 */ 9, 1, 7, 1, 1, 1, 1, 1, 1, 1, /* 20 */ 2, 2, 1);

DECLARE LB_STATE (98) FIXED BIN15 INITIAL (/* 0 */ 1, 0, 3, 45, 124, 168
   , 0, 3, 0, 7, /* 10 */ 0, 16, 29, 67, 107, 111, 116, 136, 139, 140
   , /* 20 */ 157, 0, 29, 111, 0, 31, 0, 69, 76, 77, /* 30 */ 110, 115, 0
   , 31, 0, 21, 24, 39, 40, 42, /* 40 */ 43, 46, 48, 56, 88, 123, 145, 146
   , 150, 167, /* 50 */ 69, 76, 77, 110, 115, 158, 0, 15, 33, 126
   , /* 60 */ 151, 4, 26, 86, 149, 172, 0, 15, 0, 44, /* 70 */ 100, 101
   , 102, 104, 119, 135, 0, 23, 0, 23, /* 80 */ 0, 90, 0, 91, 0, 51, 0, 92
   , 0, 129, /* 90 */ 0, 55, 96, 0, 53, 152, 0, 97, 0);

DECLARE RESUME_STATE (98) FIXED BIN15 INITIAL (/* 0 */ 513, 32, 771, 86
   , 149, 172, 769, 518, 517, 522, /* 10 */ 523, 35, 527, 528, 543, 529
   , 143, 155, 530, 531, /* 20 */ 544, 173, 67, 139, 140, 71, 75, 108, 117
   , 118, /* 30 */ 546, 142, 547, 540, 541, 775, 64, 785, 80, 786
   , /* 40 */ 83, 87, 89, 99, 125, 795, 797, 798, 163, 171, /* 50 */ 791
   , 791, 791, 791, 791, 791, 787, 569, 570, 585, /* 60 */ 586, 559, 559
   , 559, 559, 559, 558, 774, 560, 84, /* 70 */ 131, 132, 133, 134, 144
   , 154, 170, 62, 103, 595, /* 80 */ 596, 792, 776, 603, 602, 605, 604
   , 793, 777, 796, /* 90 */ 778, 790, 794, 779, 96, 96, 55, 622, 621);

/* THE SYMBOL ACCESSING THE STATES. */

DECLARE SYM_BEFORE_READ (173) FIXED BIN15 INITIAL (/* 0 */ 0, 1, 50, 52
   , 53, 55, 56, 3, 70, 76, /* 10 */ 77, 78, 79, 20, 80, 81, 72, 5, 25, 26
   , /* 20 */ 27, 32, 33, 85, 21, 34, 53, 5, 57, 5, /* 30 */ 59, 8, 51, 82
   , 74, 61, 8, 13, 14, 24, /* 40 */ 8, 16, 24, 8, 8, 2, 29, 5, 31, 69
   , /* 50 */ 89, 38, 92, 94, 95, 96, 8, 5, 48, 5, /* 60 */ 86, 35, 87, 5
   , 69, 4, 6, 62, 10, 8, /* 70 */ 15, 63, 64, 5, 19, 63, 8, 8, 69, 26
   , /* 80 */ 69, 18, 69, 69, 83, 69, 53, 69, 24, 69, /* 90 */ 36, 37, 93
   , 42, 39, 40, 96, 98, 95, 69, /* 100 */ 8, 8, 8, 87, 8, 35, 11, 8, 66
   , 69, /* 110 */ 8, 9, 4, 9, 4, 8, 9, 66, 66, 8, /* 120 */ 9, 9, 9, 4, 2
   , 69, 2, 89, 92, 42, /* 130 */ 95, 83, 83, 83, 83, 8, 8, 65, 68, 62
   , /* 140 */ 62, 5, 66, 61, 83, 24, 24, 24, 69, 53, /* 150 */ 30, 84, 94
   , 9, 83, 61, 9, 4, 4, 69, /* 160 */ 26, 69, 5, 69, 9, 11, 8, 8, 2, 8
   , /* 170 */ 83, 69, 53, 61);
DECLARE SYM_BEFORE_LA (31) FIXED BIN15 INITIAL (/* 0 */ 52, 53
   , /* 770 */ 32, 53, 57, 59, 74, 69, 89, 92, 94, 95, /* 780 */ 5, 48, 5
   , 5, 19, 69, 69, 69, 39, 40, /* 790 */ 95, 69, 89, 92, 95, 69, 94, 69
   , 69, 5);

/* END OF SLR(1) PARSING TABLES . */
```

4) SYNTHESIS PROCEDURES AND ERROR REPAIR TABLES GENERATED BY THE SLR(1) LANGUAGE ANALYZER.

(Detailed description is given in Appendix A.)

4) SYNTHESIS PROCEDURES AND ERROR REPAIR TABLES GENERATED BY THE SLR(1) LANGUAGE ANALYZER.

(Detailed description is given in Appendix A.)

```
/* S E M A N T I C S */

SYNTHESIS:
   PROCEDURE(PROD);
      DECLARE PROD FIXED;


SYN0:
   PROCEDURE(PROD_NO);
      DECLARE PROD_NO FIXED;
      DECLARE(PROD,I,OFFSET,INITIAL_PP,INITIAL_DP1,DP1_LIMIT)FIXED;

      DO CASE PROD_NO;

         CASEE0(0):   /*   NOT USED   */
            ;

         CASEE(1):
            /* <PROGRAM> ::= <MAIN BODY>                                    */
            CALL MAIN_PROG;

         CASEE(2):
            /* <MAIN BODY> ::= <DECLARATION LIST> <STATEMENT LIST>          */
            G1(MP) = G1(SP);

         CASEE(3):
            /* <MAIN BODY> ::= <DECLARATION LIST>                           */
            G1(MP) = NODE1(NOP);

         CASEE(4):
            /* <MAIN BODY> ::= <STATEMENT LIST>                             */
            ;
         CASEE(5):
            /* <DECLARATION LIST> ::= <DECLARATION>                         */
            ;

         CASEE(6):
            /* <DECLARATION LIST> ::= <DECLARATION LIST> <DECLARATION>      */
            ;

         CASEE(7):
            /* <DECLARATION> ::= <DECLARATION STATEMENT> ;                  */
            ;

         CASEE(8):
            /* <DECLARATION> ::= <PROCEDURE DEFINITION> ;                   */
            ;
```

```
/* S E M A N T I C S */

SYNTHESIS$
   PROCEDURE(PROD);
      DECLARE PROD FIXED;


SYN0$
   PROCEDURE(PROD_NO);
      DECLARE PROD_NO FIXED;
      DECLARE(PROD,I,OFFSET,INITIAL_PP,INITIAL_DP1,DP1_LIMIT)FIXED;

      DO CASE PROD_NO;

         CASEE0(0)$   /*   NOT USED   */
            ;

         CASEE(1)$
            /* <PROGRAM> $$= <MAIN BODY>                                      */
            CALL MAIN_PROG;

         CASEE(2)$
            /* <MAIN BODY> $$= <DECLARATION LIST> <STATEMENT LIST>            */
            G1(MP) = G1(SP);

         CASEE(3)$
            /* <MAIN BODY> $$= <DECLARATION LIST>                             */
            G1(MP) = NODE1(NOP);

         CASEE(4)$
            /* <MAIN BODY> $$= <STATEMENT LIST>                               */
            ;

         CASEE(5)$
            /* <DECLARATION LIST> $$= <DECLARATION>                           */
            ;

         CASEE(6)$
            /* <DECLARATION LIST> $$= <DECLARATION LIST> <DECLARATION>        */
            ;

         CASEE(7)$
            /* <DECLARATION> $$= <DECLARATION STATEMENT> ;                    */
            ;

         CASEE(8)$
            /* <DECLARATION> $$= <PROCEDURE DEFINITION> ;                     */
            ;
```

```
CASEE(19)$
   /* <IDENTIFIER SPECIFICATION> $$= ( <BIT FIELD LIST> )          */
   /*                                <DIMENSION> <TYPE>             */
   INITIAL_TYPE = TYPE(SP);
   IF TYPE(SP) = FIXEDTYPE THENN
      CALL ALIGN;
      INITIAL_START = PP; /* LOWER LIMIT */
      INITIAL_PP = PP;
      INITIAL_DP1 = DP1;
      OFFSET = 0;
      PARALLEL_ALLOCATED = FALSE;
      DO I = 0 TO ID_LIST_PTR;
         IF OFFSET + ID_WIDTH(I) > 60 THENN
            OFFSET = 0; /* ALLOCATE NEW BLOCK */
            INITIAL_PP = PP;
            INITIAL_DP1 = DP1;
            PARALLEL_ALLOCATED = FALSE;
         ELSEE
            PP = INITIAL_PP;
            DP1 = INITIAL_DP1;
         END;
         CALL ENTER(ID_STACK(I),ARRAY_DCL,TYPE(SP),VAL(SP-1),
            OFFSET,ID_WIDTH(I));
         PARALLEL_ALLOCATED = TRUE;
         OFFSET = OFFSET + ID_WIDTH(I);
      END;
      DP1_LIMIT = DP1;
      INITIAL_LIMIT = PP; /* UPPER LIMIT FOR INITIAL VALUES */
   ELSEE
      CALL ERROR('FIXED TYPE REQUIRED');
   END;
   INITIAL_PRECISION = VAL(SP); /* NUMBER OF BITS PER ELEMENT */
   PARALLEL_ALLOCATED = FALSE;

CASEE(20)$
   /* <IDENTIFIER SPECIFICATION> $$= <IDENTIFIER> ENTRY RETURNS ( */
   /*                                <TYPE> )                      */
   CALL ENTER_FORWARD_DCL(-1);

CASEE(21)$
   /* <IDENTIFIER SPECIFICATION> $$= <IDENTIFIER> ENTRY (          */
   /*                                <TYPE LIST> ) RETURNS (       */
   /*                                <TYPE> )                      */
   CALL ENTER_FORWARD_DCL(ID_LIST_PTR);

CASEE(22)$
   /* <TYPE> $$= FIXED                                            */
   TYPE(MP) = FIXEDTYPE;
   VAL(MP) = 60;

CASEE(23)$
   /* <TYPE> $$= CHARACTER ( <CONSTANT VALUE> )                   */
   TYPE(MP) = CHRTYPE;
   CALL CHECK_CONSTANT;
```

```
CASEE(19)$
   /* <IDENTIFIER SPECIFICATION> $$= ( <BIT FIELD LIST> )           */
   /*                                <DIMENSION> <TYPE>             */
   INITIAL_TYPE = TYPE(SP);
   IF TYPE(SP) = FIXEDTYPE THENN
      CALL ALIGN;
      INITIAL_START = PP; /* LOWER LIMIT */
      INITIAL_PP = PP;
      INITIAL_DP1 = DP1;
      OFFSET = 0;
      PARALLEL_ALLOCATED = FALSE;
      DO I = 0 TO ID_LIST_PTR;
         IF OFFSET + ID_WIDTH(I) > 60 THENN
            OFFSET = 0; /* ALLOCATE NEW BLOCK */
            INITIAL_PP = PP;
            INITIAL_DP1 = DP1;
            PARALLEL_ALLOCATED = FALSE;
         ELSEE
            PP = INITIAL_PP;
            DP1 = INITIAL_DP1;
         END;
         CALL ENTER(ID_STACK(I),ARRAY_DCL,TYPE(SP),VAL(SP-1),
            OFFSET,ID_WIDTH(I));
         PARALLEL_ALLOCATED = TRUE;
         OFFSET = OFFSET + ID_WIDTH(I);
      END;
      DP1_LIMIT = DP1;
      INITIAL_LIMIT = PP; /* UPPER LIMIT FOR INITIAL VALUES */
   ELSEE
      CALL ERROR('FIXED TYPE REQUIRED');
   END;
   INITIAL_PRECISION = VAL(SP); /* NUMBER OF BITS PER ELEMENT */
   PARALLEL_ALLOCATED = FALSE;

CASEE(20)$
   /* <IDENTIFIER SPECIFICATION> $$= <IDENTIFIER> ENTRY RETURNS ( */
   /*                                <TYPE> )                       */
   CALL ENTER_FORWARD_DCL(-1);

CASEE(21)$
   /* <IDENTIFIER SPECIFICATION> $$= <IDENTIFIER> ENTRY (          */
   /*                                <TYPE LIST> ) RETURNS (        */
   /*                                <TYPE> )                       */
   CALL ENTER_FORWARD_DCL(ID_LIST_PTR);

CASEE(22)$
   /* <TYPE> $$= FIXED                                              */
   TYPE(MP) = FIXEDTYPE;
   VAL(MP) = 60;

CASEE(23)$
   /* <TYPE> $$= CHARACTER ( <CONSTANT VALUE> )                     */
   TYPE(MP) = CHRTYPE;
   CALL CHECK_CONSTANT;
```

```
CASEE(24)$
   /* <TYPE> $$= BIT ( <CONSTANT VALUE> )                         */
   TYPE(MP) = CHRTYPE;
   CALL CHECK_CONSTANT;

CASEE(25)$
   /* <DIMENSION> $$= ( <CONSTANT VALUE> )                        */
   CALL CHECK_CONSTANT;

CASEE(26)$
   /* <IDENTIFIER LIST> $$= <IDENTIFIER>                          */
   ID_LIST_PTR = 0;
   ID_STACK(0) = VAR(MP);

CASEE(27)$
   /* <IDENTIFIER LIST> $$= <IDENTIFIER LIST> , <IDENTIFIER>      */
   INC_ID_PTR;
   ID_STACK(ID_LIST_PTR) = VAR(SP);

CASEE(28)$
   /* <BIT FIELD LIST> $$= <BIT SPECIFICATION>                    */
   ID_LIST_PTR = 0;
   ID_STACK(0) = VAR(MP);
   ID_WIDTH(0) = VAL(MP);

CASEE(29)$
   /* <BIT FIELD LIST> $$= <BIT FIELD LIST> , <BIT SPECIFICATION> */
   INC_ID_PTR;
   ID_STACK(ID_LIST_PTR) = VAR(SP);
   ID_WIDTH(ID_LIST_PTR) = VAL(SP);

CASEE(30)$
   /* <BIT SPECIFICATION> $$= <IDENTIFIER> ( <CONSTANT VALUE> )   */
   CALL CHECK_CONSTANT;
   IF VAL(MP) > 60 THENN
      CALL ERROR('PRECISION TOO BIG');
      VAL(SP) = 60;
   END;
   VAL(MP) = VAL(SP);

CASEE(31)$
   /* <TYPE LIST> $$= <TYPE>                                      */
   ID_LIST_PTR = 0;
   ID_TYPE(0) = TYPE(MP);
   ID_WIDTH(0) = VAL(MP);

CASEE(32)$
   /* <TYPE LIST> $$= <TYPE LIST> , <TYPE>                        */
   INC_ID_PTR;
   ID_TYPE(ID_LIST_PTR)= TYPE(SP);
   ID_WIDTH(ID_LIST_PTR) = VAL(SP);
```

```
CASEE(24)$
   /* <TYPE> $$= BIT ( <CONSTANT VALUE> )                            */
   TYPE(MP) = CHRTYPE;
   CALL CHECK_CONSTANT;

CASEE(25)$
   /* <DIMENSION> $$= ( <CONSTANT VALUE> )                           */
   CALL CHECK_CONSTANT;

CASEE(26)$
   /* <IDENTIFIER LIST> $$= <IDENTIFIER>                             */
   ID_LIST_PTR = 0;
   ID_STACK(0) = VAR(MP);

CASEE(27)$
   /* <IDENTIFIER LIST> $$= <IDENTIFIER LIST> , <IDENTIFIER>         */
   INC_ID_PTR;
   ID_STACK(ID_LIST_PTR) = VAR(SP);

CASEE(28)$
   /* <BIT FIELD LIST> $$= <BIT SPECIFICATION>                       */
   ID_LIST_PTR = 0;
   ID_STACK(0) = VAR(MP);
   ID_WIDTH(0) = VAL(MP);

CASEE(29)$
   /* <BIT FIELD LIST> $$= <BIT FIELD LIST> , <BIT SPECIFICATION> */
   INC_ID_PTR;
   ID_STACK(ID_LIST_PTR) = VAR(SP);
   ID_WIDTH(ID_LIST_PTR) = VAL(SP);

CASEE(30)$
   /* <BIT SPECIFICATION> $$= <IDENTIFIER> ( <CONSTANT VALUE> )   */
   CALL CHECK_CONSTANT;
   IF VAL(MP) > 60 THENN
      CALL ERROR('PRECISION TOO BIG');
      VAL(SP) = 60;
   END;
   VAL(MP) = VAL(SP);

CASEE(31)$
   /* <TYPE LIST> $$= <TYPE>                                         */
   ID_LIST_PTR = 0;
   ID_TYPE(0) = TYPE(MP);
   ID_WIDTH(0) = VAL(MP);

CASEE(32)$
   /* <TYPE LIST> $$= <TYPE LIST> , <TYPE>                           */
   INC_ID_PTR;
   ID_TYPE(ID_LIST_PTR)= TYPE(SP);
   ID_WIDTH(ID_LIST_PTR) = VAL(SP);
```

```
CASEE(33)%
   /* <INITIAL LIST> %%= INITIAL ( <CONSTANT LIST> )                        */
   ;

CASEE(34)%
   /* <CONSTANT LIST> %%= <CONSTANT VALUE>                                   */
   IF LEVELNO > 0 THENN
      CALL ERROR('INITIAL VALUES ILLEGAL IN RECURSIVE PROCEDURE');
   END;
   PP = INITIAL_START;
   CALL SETINIT(MP);

CASEE(35)%
   /* <CONSTANT LIST> %%= <CONSTANT LIST> , <CONSTANT VALUE>                 */
   CALL SETINIT(SP);

CASEE(36)%
   /* <CONSTANT VALUE> %%= <EXPRESSION>                                      */
   IF CLASS(MP) NE CONSTANT_CLASS THENN
      CALL ERROR('ONLY CONSTANT EXPRESSIONS ALLOWED');
      VAL(MP) = 0;
      VAR(MP) = EMPTY_STRING_PTR;
   END;

CASEE(37)%
   /* <PROCEDURE DEFINITION> %%= <PROCEDURE HEAD> <MAIN BODY>                */
   /*                            <ENDING>                                    */
   CALL CLOSE_SCOPE;

CASEE(38)%
   /* <PROCEDURE HEAD> %%= <PROCEDURE NAME> ;                                */
   CALL ENTER_PROC(VAR(MP),VAL(MP),-1,FIXEDTYPE);

CASEE(39)%
   /* <PROCEDURE HEAD> %%= <PROCEDURE NAME> <TYPE> ;                         */
   CALL ENTER_PROC(VAR(MP),VAL(MP),-1,TYPE(SP-1));

CASEE(40)%
   /* <PROCEDURE HEAD> %%= <PROCEDURE NAME> ( <IDENTIFIER LIST> ) */
   /*                      ;                                      */
   CALL ENTER_PROC(VAR(MP),VAL(MP),ID_LIST_PTR,
      FIXEDTYPE);

CASEE(41)%
   /* <PROCEDURE HEAD> %%= <PROCEDURE NAME> ( <IDENTIFIER LIST> ) */
   /*                      <TYPE> ;                               */
   CALL ENTER_PROC(VAR(MP),VAL(MP),ID_LIST_PTR,
      TYPE(SP-1));

CASEE(42)%
   /* <PROCEDURE NAME> %%= <IDENTIFIER> % PROCEDURE                          */
   VAL(MP) = PROC_DCL;
```

```
CASEE(33)%
   /* <INITIAL LIST> %%= INITIAL ( <CONSTANT LIST> )                    */
   ;

CASEE(34)%
   /* <CONSTANT LIST> %%= <CONSTANT VALUE>                              */
   IF LEVELNO > 0 THENN
      CALL ERROR('INITIAL VALUES ILLEGAL IN RECURSIVE PROCEDURE');
   END;
   PP = INITIAL_START;
   CALL SETINIT(MP);

CASEE(35)%
   /* <CONSTANT LIST> %%= <CONSTANT LIST> , <CONSTANT VALUE>            */
   CALL SETINIT(SP);

CASEE(36)%
   /* <CONSTANT VALUE> %%= <EXPRESSION>                                 */
   IF CLASS(MP) NE CONSTANT_CLASS THENN
      CALL ERROR('ONLY CONSTANT EXPRESSIONS ALLOWED');
      VAL(MP) = 0;
      VAR(MP) = EMPTY_STRING_PTR;
   END;

CASEE(37)%
   /* <PROCEDURE DEFINITION> %%= <PROCEDURE HEAD> <MAIN BODY>           */
   /*                            <ENDING>                               */
   CALL CLOSE_SCOPE;

CASEE(38)%
   /* <PROCEDURE HEAD> %%= <PROCEDURE NAME> ;                           */
   CALL ENTER_PROC(VAR(MP),VAL(MP),-1,FIXEDTYPE);

CASEE(39)%
   /* <PROCEDURE HEAD> %%= <PROCEDURE NAME> <TYPE> ;                    */
   CALL ENTER_PROC(VAR(MP),VAL(MP),-1,TYPE(SP-1));

CASEE(40)%
   /* <PROCEDURE HEAD> %%= <PROCEDURE NAME> ( <IDENTIFIER LIST> ) */
   /*                                       ;                     */
   CALL ENTER_PROC(VAR(MP),VAL(MP),ID_LIST_PTR,
      FIXEDTYPE);

CASEE(41)%
   /* <PROCEDURE HEAD> %%= <PROCEDURE NAME> ( <IDENTIFIER LIST> ) */
   /*                      <TYPE> ;                               */
   CALL ENTER_PROC(VAR(MP),VAL(MP),ID_LIST_PTR,
      TYPE(SP-1));

CASEE(42)%
   /* <PROCEDURE NAME> %%= <IDENTIFIER> % PROCEDURE                     */
   VAL(MP) = PROC_DCL;
```

```
      CASEE(43)%
         /* <PROCEDURE NAME> %%= <IDENTIFIER> % RECURSIVE PROCEDURE     */
         VAL(MP) = RECURSIVE_PROC_DCL;

      CASEE(44)%
         /* <ENDING> %%= END_PROC                                       */
         VAR(MP) = EMPTY_STRING_PTR;

    END_CASE;

  END SYN0;

SYN1%
  PROCEDURE(PROD_NO);
    DECLARE PROD_NO FIXED;
    DECLARE(PROD,I,J,K)FIXED;

    DO CASE PROD_NO;

      CASEE0(45)%
         /* <ENDING> %%= END_PROC <IDENTIFIER>                          */
         VAR(MP) = VAR(SP);

      CASEE(46)%
         /* <STATEMENT LIST> %%= <STATEMENT>                            */
         ;

      CASEE(47)%
         /* <STATEMENT LIST> %%= <STATEMENT LIST> <STATEMENT>           */
         G1(MP) = LINK_CHAIN(G1(MP),G1(SP));

      CASEE(48)%
         /* <STATEMENT> %%= <BASIC STATEMENT>                           */
         ;

      CASEE(49)%
         /* <STATEMENT> %%= <IF STATEMENT>                              */
         ;

      CASEE(50)%
         /* <BASIC STATEMENT> %%= <ASSIGNMENT> ;                        */
         ;

      CASEE(51)%
         /* <BASIC STATEMENT> %%= <GROUP> ;                             */
         ;

      CASEE(52)%
         /* <BASIC STATEMENT> %%= <RETURN STATEMENT> ;                  */
         ;

      CASEE(53)%
         /* <BASIC STATEMENT> %%= <CALL STATEMENT> ;                    */
         ;
      CASEE(43)$
         /* <PROCEDURE NAME> $$= <IDENTIFIER> $ RECURSIVE PROCEDURE      */
         VAL(MP) = RECURSIVE_PROC_DCL;

      CASEE(44)$
         /* <ENDING> $$= END_PROC                                        */
         VAR(MP) = EMPTY_STRING_PTR;

   END_CASE;

END SYNO;

SYN1$
   PROCEDURE(PROD_NO);
      DECLARE PROD_NO FIXED;
      DECLARE(PROD,I,J,K)FIXED;

      DO CASE PROD_NO;

         CASEE0(45)$
          /* <ENDING> $$= END_PROC <IDENTIFIER>                             */
          VAR(MP) = VAR(SP);

         CASEE(46)$
            /* <STATEMENT LIST> $$= <STATEMENT>                             */
            ;

         CASEE(47)$
            /* <STATEMENT LIST> $$= <STATEMENT LIST> <STATEMENT>            */
            G1(MP) = LINK_CHAIN(G1(MP),G1(SP));

         CASEE(48)$
            /* <STATEMENT> $$= <BASIC STATEMENT>                            */
            ;

         CASEE(49)$
            /* <STATEMENT> $$= <IF STATEMENT>                               */
            ;

         CASEE(50)$
            /* <BASIC STATEMENT> $$= <ASSIGNMENT> ;                         */
            ;

         CASEE(51)$
            /* <BASIC STATEMENT> $$= <GROUP> ;                              */
            ;

         CASEE(52)$
            /* <BASIC STATEMENT> $$= <RETURN STATEMENT> ;                   */
            ;

         CASEE(53)$
            /* <BASIC STATEMENT> $$= <CALL STATEMENT> ;                     */
            ;
```

```
CASEE(54)%
   /* <BASIC STATEMENT> %%= EXIT ;                                  */
   ;

CASEE(55)%
   /* <BASIC STATEMENT> %%= <GENERATE BLOCK> ;                      */
   ;

CASEE(56)%
   /* <BASIC STATEMENT> %%= ;                                       */
   G1(MP) = SHORT_STATEMENT(NOP,-1,CARDNUM(MP));

CASEE(57)%
   /* <IF STATEMENT> %%= <IF CLAUSE> <STATEMENT>                    */
   IF CLASS(MP) = CONSTANT_CLASS THENN
      IF VAL(MP) = 1 THENN
         G1(MP) = G1(SP);
      END;
   ELSEE
      G1(MP) = IFTHENELSE(G1(MP),NOP,G1(SP),CARDNUM(MP));
   END;

CASEE(58)%
   /* <IF STATEMENT> %%= <IF CLAUSE> <TRUE PART> <STATEMENT>        */
   IF CLASS(MP) = CONSTANT_CLASS THENN
      IF VAL(MP) = 1 THENN
         G1(MP) = G1(MPP1);
      ELSEE
         G1(MP) = G1(SP);
      END;
   ELSEE
      G1(MP) = IFTHENELSE(G1(MP),G1(SP),G1(MPP1),CARDNUM(MP));
   END;

CASEE(59)%
   /* <IF CLAUSE> %%= IF <EXPRESSION> THEN                          */
   CALL MOVE_STACKS(MPP1,MP);

CASEE(60)%
   /* <TRUE PART> %%= <BASIC STATEMENT> ELSE                        */
   ;

CASEE(61)%
   /* <ASSIGNMENT> %%= <IDENTIFIER> = <EXPRESSION>                  */
   CALL GEN_STORE(FALSE);

CASEE(62)%
   /* <ASSIGNMENT> %%= <IDENTIFIER> ( <EXPRESSION> ) =              */
   /*                  <EXPRESSION>                                 */
   CALL GEN_STORE(TRUE);
```

```
CASEE(54)%
   /* <BASIC STATEMENT> ::= EXIT ;                                      */
   ;

CASEE(55)%
   /* <BASIC STATEMENT> ::= <GENERATE BLOCK> ;                          */
   ;

CASEE(56)%
   /* <BASIC STATEMENT> ::= ;                                           */
   G1(MP) = SHORT_STATEMENT(NOP,-1,CARDNUM(MP));

CASEE(57)%
   /* <IF STATEMENT> ::= <IF CLAUSE> <STATEMENT>                        */
   IF CLASS(MP) = CONSTANT_CLASS THENN
      IF VAL(MP) = 1 THENN
         G1(MP) = G1(SP);
      END;
   ELSEE
      G1(MP) = IFTHENELSE(G1(MP),NOP,G1(SP),CARDNUM(MP));
   END;

CASEE(58)%
   /* <IF STATEMENT> ::= <IF CLAUSE> <TRUE PART> <STATEMENT>            */
   IF CLASS(MP) = CONSTANT_CLASS THENN
      IF VAL(MP) = 1 THENN
         G1(MP) = G1(MPP1);
      ELSEE
         G1(MP) = G1(SP);
      END;
   ELSEE
      G1(MP) = IFTHENELSE(G1(MP),G1(SP),G1(MPP1),CARDNUM(MP));
   END;

CASEE(59)%
   /* <IF CLAUSE> ::= IF <EXPRESSION> THEN                              */
   CALL MOVE_STACKS(MPP1,MP);

CASEE(60)%
   /* <TRUE PART> ::= <BASIC STATEMENT> ELSE                            */
   ;

CASEE(61)%
   /* <ASSIGNMENT> ::= <IDENTIFIER> = <EXPRESSION>                      */
   CALL GEN_STORE(FALSE);

CASEE(62)%
   /* <ASSIGNMENT> ::= <IDENTIFIER> ( <EXPRESSION> ) =                  */
   /*                  <EXPRESSION>                                     */
   CALL GEN_STORE(TRUE);
```

```
CASEE(63)%
   /* <ASSIGNMENT> %%= OUTPUT = <EXPRESSION>                              */
   IF TYPE(SP) = FIXEDTYPE THENN
      CALL EXP_CONVERT(SP);
   END;
   G1(MP) = LONG_STATEMENT(71,-1,FORCE_GRAPH(SP),0,0,CARDNUM(MP));

CASEE(64)%
   /* <ASSIGNMENT> %%= OUTPUT ( <EXPRESSION> ) = <EXPRESSION>             */
   IF TYPE(SP) = FIXEDTYPE THENN
      CALL EXP_CONVERT(SP);
   END;
   G1(MP) = LONG_STATEMENT(71,FORCE_GRAPH(MP+2),
                           FORCE_GRAPH(SP),1,0,CARDNUM(MP));

CASEE(65)%
   /* <ASSIGNMENT> %%= FILE ( <EXPRESSION LIST> ) = <IDENTIFIER>  */
   EXP_CNT = FIXL(MP+2);
   IF CNT(MP+2) = 2 THENN /* 2 EXPRESSIONS */
      K = ID_LOOKUP(SP);
      CALL EXPAND_EXP(EXP_CNT+1,MP);
      CALL EXPAND_EXP(EXP_CNT+2,MPP1);
      IF TYPE(MP) = FIXEDTYPE AND TYPE(MPP1) = FIXEDTYPE THENN
         G1(MP) = EXPRESSION_LIST(FORCE_GRAPH(MP),
                                  FORCE_GRAPH(MPP1),-1);
         G1(MP) = LONG_STATEMENT(75,G1(MP),NODE2(LA,K),0,0,
                                                 CARDNUM(MP));
      ELSEE
         CALL ERROR('FILE PARAMETERS MUST BE FIXED TYPE');
      END;
   ELSEE
      CALL ERROR('FILE NEEDS TWO PARAMETERS');
   END;

CASEE(66)%
   /* <ASSIGNMENT> %%= FILE ( <EXPRESSION LIST> ) = <IDENTIFIER>  */
   /*                 ( <EXPRESSION> )                             */
   ;

CASEE(67)%
   /* <ASSIGNMENT> %%= <IDENTIFIER> = FILE ( <EXPRESSION LIST> )  */
   EXP_CNT = FIXL(SP-1);
   IF CNT(SP-1) = 2 THENN /* 2 EXPRESSIONS */
      K = ID_LOOKUP(MP);
      CALL EXPAND_EXP(EXP_CNT+1,MPP1);
      CALL EXPAND_EXP(EXP_CNT+2,SP);
      IF TYPE(MPP1) = FIXEDTYPE AND TYPE(SP) = FIXEDTYPE THENN
         G1(MP) = EXPRESSION_LIST(FORCE_GRAPH(MPP1),FORCE_GRAPH(
            SP),-1);
         G1(MP) = LONG_STATEMENT(74,G1(MP),NODE2(LA,K),0,0,
                                                 CARDNUM(MP));
      ELSEE
         CALL ERROR('FILE PARAMETERS MUST BE FIXED TYPE');
      END;
```

```
CASEE(63)%
   /* <ASSIGNMENT> %%= OUTPUT = <EXPRESSION>                          */
   IF TYPE(SP) = FIXEDTYPE THENN
      CALL EXP_CONVERT(SP);
   END;
   G1(MP) = LONG_STATEMENT(71,-1,FORCE_GRAPH(SP),0,0,CARDNUM(MP));

CASEE(64)%
   /* <ASSIGNMENT> %%= OUTPUT ( <EXPRESSION> ) = <EXPRESSION>        */
   IF TYPE(SP) = FIXEDTYPE THENN
      CALL EXP_CONVERT(SP);
   END;
   G1(MP) = LONG_STATEMENT(71,FORCE_GRAPH(MP+2),
                           FORCE_GRAPH(SP),1,0,CARDNUM(MP));

CASEE(65)%
   /* <ASSIGNMENT> %%= FILE ( <EXPRESSION LIST> ) = <IDENTIFIER>   */
   EXP_CNT = FIXL(MP+2);
   IF CNT(MP+2) = 2 THENN /* 2 EXPRESSIONS */
      K = ID_LOOKUP(SP);
      CALL EXPAND_EXP(EXP_CNT+1,MP);
      CALL EXPAND_EXP(EXP_CNT+2,MPP1);
      IF TYPE(MP) = FIXEDTYPE AND TYPE(MPP1) = FIXEDTYPE THENN
         G1(MP) = EXPRESSION_LIST(FORCE_GRAPH(MP),
                                  FORCE_GRAPH(MPP1),-1);
         G1(MP) = LONG_STATEMENT(75,G1(MP),NODE2(LA,K),0,0,
                                                CARDNUM(MP));
      ELSEE
         CALL ERROR('FILE PARAMETERS MUST BE FIXED TYPE');
      END;
   ELSEE
      CALL ERROR('FILE NEEDS TWO PARAMETERS');
   END;

CASEE(66)%
   /* <ASSIGNMENT> %%= FILE ( <EXPRESSION LIST> ) = <IDENTIFIER>   */
   /*                ( <EXPRESSION> )                              */
   ;

CASEE(67)%
   /* <ASSIGNMENT> %%= <IDENTIFIER> = FILE ( <EXPRESSION LIST> )   */
   EXP_CNT = FIXL(SP-1);
   IF CNT(SP-1) = 2 THENN /* 2 EXPRESSIONS */
      K = ID_LOOKUP(MP);
      CALL EXPAND_EXP(EXP_CNT+1,MPP1);
      CALL EXPAND_EXP(EXP_CNT+2,SP);
      IF TYPE(MPP1) = FIXEDTYPE AND TYPE(SP) = FIXEDTYPE THENN
         G1(MP) = EXPRESSION_LIST(FORCE_GRAPH(MPP1),FORCE_GRAPH(
            SP),-1);
         G1(MP) = LONG_STATEMENT(74,G1(MP),NODE2(LA,K),0,0,
                                                CARDNUM(MP));
      ELSEE
         CALL ERROR('FILE PARAMETERS MUST BE FIXED TYPE');
      END;
   ELSEE
      CALL ERROR('FILE NEEDS TWO PARAMETERS');
   END;

CASEE(68)%
   /* <ASSIGNMENT> %%= <IDENTIFIER> ( <EXPRESSION> ) = FILE (      */
   /*                     <EXPRESSION LIST> )                      */
   ;

CASEE(69)%
   /* <GROUP> %%= DO ; <STATEMENT LIST> END                        */
   G1(MP) = G1(MP+2);

CASEE(70)%
   /* <GROUP> %%= DO WHILE <EXPRESSION> ; <STATEMENT LIST> END     */
   G1(MP) = DOWHILE(FORCE_GRAPH(MP+2),G1(SP-1),NOP,CARDNUM(MP));

CASEE(71)%
   /* <GROUP> %%= DO <IDENTIFIER> = <EXPRESSION> TO <EXPRESSION>   */
   /*             ; <STATEMENT LIST> END                           */
   I = ID_LOOKUP(MPP1);
   G1(MP) = LONG_STATEMENT(ST,-1,FORCE_GRAPH(MP+3),I,0,
                      CARDNUM(MP) );
   J = EMIT_WORD(0);
   G1(MPP1) = LONG_STATEMENT(STW,-1,FORCE_GRAPH(MP+5),
                      0,SHR(J,2),CARDNUM(MP));
   G1(MP) = LINK_CHAIN(G1(MP),G1(MPP1));
   G1(MPP1)=BINARY_EXPRESSION(23,NODE2(L,I),NODE3(LW,0,SHR(J,2))
         ,0,0);
   G1(SP) = BINARY_EXPRESSION(ADD,NODE2(L,I),NODE2(LIT,1),0,0);
   G1(SP) = LONG_STATEMENT(ST,-1,G1(SP),I,0,CARDNUM(MP));
   G1(MPP1) = DOWHILE(G1(MPP1),G1(SP-1),G1(SP),CARDNUM(MP));
   G1(MP) = LINK_CHAIN(G1(MP),G1(MPP1));

CASEE(72)%
   /* <GROUP> %%= DO CASE <EXPRESSION> ; <CASE BODY> END           */
   G1(MP) = DOCASE(FORCE_GRAPH(MP+2),G1(SP-1),NOP,CARDNUM(MP));
   INFORMATION = '';

CASEE(73)%
   /* <CASE BODY> %%= <STATEMENT>                                  */
   CNT(MP) = 1; /* FIRST CASE */
   INFORMATION = ' CASE 1';
   G1(MP) = SHORT_STATEMENT(NOP,G1(MP),CARDNUM(MP));

CASEE(74)%
   /* <CASE BODY> %%= <CASE BODY> <STATEMENT>                      */
   CNT(MP) = CNT(MP) + 1;
   INFORMATION = ' CASE '||CNT(MP);
   G1(SP) = SHORT_STATEMENT(NOP,G1(SP),CARDNUM(MP));
   G1(MP) = LINK_CHAIN(G1(MP),G1(SP));

CASEE(75)%
   /* <RETURN STATEMENT> %%= RETURN                                */
   G1(MP) = SPECIAL(RTRN,PROC_INDEX,CARDNUM(MP));
   ELSEE
      CALL ERROR('FILE NEEDS TWO PARAMETERS');
   END;

CASEE(68)$
   /* <ASSIGNMENT> $$= <IDENTIFIER> ( <EXPRESSION> ) = FILE (     */
   /*                  <EXPRESSION LIST> )                          */
   ;

CASEE(69)$
   /* <GROUP> $$= DO ; <STATEMENT LIST> END                         */
   G1(MP) = G1(MP+2);

CASEE(70)$
   /* <GROUP> $$= DO WHILE <EXPRESSION> ; <STATEMENT LIST> END      */
   G1(MP) = DOWHILE(FORCE_GRAPH(MP+2),G1(SP-1),NOP,CARDNUM(MP));

CASEE(71)$
   /* <GROUP> $$= DO <IDENTIFIER> = <EXPRESSION> TO <EXPRESSION>   */
   /*           ; <STATEMENT LIST> END                               */
   I = ID_LOOKUP(MPP1);
   G1(MP) = LONG_STATEMENT(ST,-1,FORCE_GRAPH(MP+3),I,0,
                          CARDNUM(MP) );
   J = EMIT_WORD(0);
   G1(MPP1) = LONG_STATEMENT(STW,-1,FORCE_GRAPH(MP+5),
                          0,SHR(J,2),CARDNUM(MP));
   G1(MP) = LINK_CHAIN(G1(MP),G1(MPP1));
   G1(MPP1)=BINARY_EXPRESSION(23,NODE2(L,I),NODE3(LW,0,SHR(J,2))
         ,0,0);
   G1(SP) = BINARY_EXPRESSION(ADD,NODE2(L,I),NODE2(LIT,1),0,0);
   G1(SP) = LONG_STATEMENT(ST,-1,G1(SP),I,0,CARDNUM(MP));
   G1(MPP1) = DOWHILE(G1(MPP1),G1(SP-1),G1(SP),CARDNUM(MP));
   G1(MP) = LINK_CHAIN(G1(MP),G1(MPP1));

CASEE(72)$
   /* <GROUP> $$= DO CASE <EXPRESSION> ; <CASE BODY> END            */
   G1(MP) = DOCASE(FORCE_GRAPH(MP+2),G1(SP-1),NOP,CARDNUM(MP));
   INFORMATION = '';

CASEE(73)$
   /* <CASE BODY> $$= <STATEMENT>                                   */
   CNT(MP) = 1; /* FIRST CASE */
   INFORMATION = ' CASE 1';
   G1(MP) = SHORT_STATEMENT(NOP,G1(MP),CARDNUM(MP));

CASEE(74)$
   /* <CASE BODY> $$= <CASE BODY> <STATEMENT>                       */
   CNT(MP) = CNT(MP) + 1;
   INFORMATION = ' CASE '||CNT(MP);
   G1(SP) = SHORT_STATEMENT(NOP,G1(SP),CARDNUM(MP));
   G1(MP) = LINK_CHAIN(G1(MP),G1(SP));

CASEE(75)$
   /* <RETURN STATEMENT> $$= RETURN                                 */
   G1(MP) = SPECIAL(RTRN,PROC_INDEX,CARDNUM(MP));
```

```
CASEE(76)$
   /* <RETURN STATEMENT> $$= RETURN <EXPRESSION>                        */
   IF RETURNED_TYPE = CHRTYPE AND TYPE(SP) = FIXEDTYPE THENN
      CALL EXP_CONVERT(SP);
   END;
   IF RETURNED_TYPE NE TYPE(SP) THENN
      CALL ERROR('ILLEGAL RETURN TYPE');
   ELSEE
      G1(MP) = LONG_STATEMENT(RTRNX,-1,FORCE_GRAPH(SP),
                              PROC_INDEX,0,CARDNUM(MP));
   END;

CASEE(77)$
   /* <CALL STATEMENT> $$= CALL <IDENTIFIER>                            */
   G1(MP) = GENERATE_CALL(ID_LOOKUP(MPP1),-1,CALLP);
   G1(MP) = SHORT_STATEMENT(NOP,G1(MP),CARDNUM(MP));

CASEE(78)$
   /* <CALL STATEMENT> $$= CALL <IDENTIFIER> ( <EXPRESSION LIST>        */
   /*                                        )                          */
   EXP_CNT = FIXL(SP-1);
   I = ID_LOOKUP(MPP1);
   IF I NE APPEND_STRING THENN
      G1(MP) = GENERATE_CALL(I,CNT(SP-1),CALLP);
      G1(MP) = SHORT_STATEMENT(NOP,G1(MP),CARDNUM(MP));
   ELSEE
      CALL GENERATE_APPEND(CNT(SP-1));
   END;

CASEE(79)$
   /* <CALL STATEMENT> $$= CALL EXIT                                    */
   G1(MP) = SPECIAL(SRTRN,0,CARDNUM(MP));

CASEE(80)$
   /* <GENERATE BLOCK> $$= <GENERATE HEAD> <INLINE BODY> END            */
   ;

CASEE(81)$
   /* <GENERATE BLOCK> $$= <GENERATE HEAD> <INLINE BODY> <LABEL>        */
   /*                      END                                          */
   ;

CASEE(82)$
   /* <GENERATE HEAD> $$= GENERATE ;                                    */
   ;

CASEE(83)$
   /* <INLINE BODY> $$= <INLINE STATEMENT>                              */
   ;

CASEE(84)$
   /* <INLINE BODY> $$= <INLINE BODY> <INLINE STATEMENT>                */
   ;
```

```
        CASEE(85)%
           /* <INLINE STATEMENT> %%= INLINE ( <EXPRESSION LIST> ) ;       */
           ;

        CASEE(86)%
           /* <INLINE STATEMENT> %%= <LABEL> INLINE ( <EXPRESSION LIST> ) */
           /*                        ;                                    */
           ;

      END_CASE;

   END SYN1;

SYN2%
  PROCEDURE(PROD_NO);
      DECLARE PROD_NO FIXED;
      DECLARE PROD_FIXED;

      DO CASE PROD_NO;

        CASEE(87)%
           /* <LABEL> %%= <IDENTIFIER> %                                  */
           ;

        CASEE(88)%
           /* <EXPRESSION> %%= <LOGICAL FACTOR>                          */
           ;

        CASEE(89)%
           /* <EXPRESSION> %%= <EXPRESSION> OR <LOGICAL FACTOR>          */
           CALL EVALUATE(41); /* OR */

        CASEE(90)%
           /* <LOGICAL FACTOR> %%= <LOGICAL SECONDARY>.                  */
           ;

        CASEE(91)%
           /* <LOGICAL FACTOR> %%= <LOGICAL FACTOR> AND                  */
           /*                      <LOGICAL SECONDARY>                   */
           CALL EVALUATE(40); /* AND */

        CASEE(92)%
           /* <LOGICAL SECONDARY> %%= <LOGICAL PRIMARY>                  */
           ;

        CASEE(93)%
           /* <LOGICAL SECONDARY> %%= NOT <LOGICAL PRIMARY>              */
           CALL EVALUATE_UNARY(42); /* NOT */

        CASEE(94)%
           /* <LOGICAL PRIMARY> %%= <STRING EXPRESSION>                  */
           ;
```

```
CASEE(95)$
   /* <LOGICAL PRIMARY> $$= <STRING EXPRESSION> <RELATION>        */
   /*                        <STRING EXPRESSION>                   */
   CALL EVALUATE(VAL(MPP1)); /* <RELATION */

CASEE(96)$
   /* <RELATION> $$= =                                             */
   VAL(MP) = 20;

CASEE(97)$
   /* <RELATION> $$= <                                             */
   VAL(MP) = 21;

CASEE(98)$
   /* <RELATION> $$= >                                             */
   VAL(MP) = 22;

CASEE(99)$
   /* <RELATION> $$= < =                                           */
   VAL(MP) = 23;

CASEE(100)$
   /* <RELATION> $$= > =                                           */
   VAL(MP) = 24;

CASEE(101)$
   /* <RELATION> $$= NE                                            */
   VAL(MP) = 25;

CASEE(102)$
   /* <STRING EXPRESSION> $$= <ARITHMETIC EXPRESSION>              */
   ;

CASEE(103)$
   /* <STRING EXPRESSION> $$= <STRING EXPRESSION> | |              */
   /*                        <ARITHMETIC EXPRESSION>               */
   CALL EVALUATE(50); /* CONCATENATE */

CASEE(104)$
   /* <ARITHMETIC EXPRESSION> $$= <TERM>                           */
   ;

CASEE(105)$
   /* <ARITHMETIC EXPRESSION> $$= <ARITHMETIC EXPRESSION> <ADD>    */
   /*                            <TERM>                            */
   CALL EVALUATE(VAL(MPP1));

CASEE(106)$
   /* <ARITHMETIC EXPRESSION> $$= <ADD> <TERM>                     */
   CALL EVALUATE_UNARY(VAL(MP));
```

```
CASEE(95)$
   /* <LOGICAL PRIMARY> $$= <STRING EXPRESSION> <RELATION>         */
   /*                       <STRING EXPRESSION>                    */
   CALL EVALUATE(VAL(MPP1)); /* <RELATION */

CASEE(96)$
   /* <RELATION> $$= =                                             */
   VAL(MP) = 20;

CASEE(97)$
   /* <RELATION> $$= <                                             */
   VAL(MP) = 21;

CASEE(98)$
   /* <RELATION> $$= >                                             */
   VAL(MP) = 22;

CASEE(99)$
   /* <RELATION> $$= < =                                           */
   VAL(MP) = 23;

CASEE(100)$
   /* <RELATION> $$= > =                                           */
   VAL(MP) = 24;

CASEE(101)$
   /* <RELATION> $$= NE                                            */
   VAL(MP) = 25;

CASEE(102)$
   /* <STRING EXPRESSION> $$= <ARITHMETIC EXPRESSION>              */
   ;

CASEE(103)$
   /* <STRING EXPRESSION> $$= <STRING EXPRESSION> | |              */
   /*                         <ARITHMETIC EXPRESSION>              */
   CALL EVALUATE(50); /* CONCATENATE */

CASEE(104)$
   /* <ARITHMETIC EXPRESSION> $$= <TERM>                           */
   ;

CASEE(105)$
   /* <ARITHMETIC EXPRESSION> $$= <ARITHMETIC EXPRESSION> <ADD>    */
   /*                             <TERM>                           */
   CALL EVALUATE(VAL(MPP1));

CASEE(106)$
   /* <ARITHMETIC EXPRESSION> $$= <ADD> <TERM>                     */
   CALL EVALUATE_UNARY(VAL(MP));
```

```
CASEE(107)$
   /* <ADD> $$= +                                                   */
   VAL(MP) = ADD;

CASEE(108)$
   /* <ADD> $$= -                                                   */
   VAL(MP) = SUBT;

CASEE(109)$
   /* <TERM> $$= <PRIMARY>                                          */
   ;

CASEE(110)$
   /* <TERM> $$= <TERM> <MULT> <PRIMARY>                            */
   CALL EVALUATE(VAL(MPP1));

CASEE(111)$
   /* <MULT> $$= *                                                  */
   VAL(MP) = MULT;

CASEE(112)$
   /* <MULT> $$= /                                                  */
   VAL(MP) = DIV;

CASEE(113)$
   /* <MULT> $$= MOD                                                */
   VAL(MP) = MODD ;

CASEE(114)$
   /* <PRIMARY> $$= <CONSTANT>                                      */
   CLASS(MP) = CONSTANT_CLASS;

CASEE(115)$
   /* <PRIMARY> $$= <VARIABLE>                                      */
   ;

CASEE(116)$
   /* <PRIMARY> $$= ( <EXPRESSION> )                                */
   CALL MOVE_STACKS(MPP1,MP);

CASEE(117)$
   /* <VARIABLE> $$= <IDENTIFIER>                                   */
   CALL ACCESS_VARIABLE(0);

CASEE(118)$
   /* <VARIABLE> $$= <IDENTIFIER> ( <EXPRESSION LIST> )             */
   EXP_CNT = FIXL(SP-1);
   CALL ACCESS_VARIABLE(CNT(SP-1));

CASEE(119)$
   /* <VARIABLE> $$= <BUILTIN FUNCTION>                             */
   CALL EVALUATE_BUILTIN_FUNCTION(0);
```

```
CASEE(107)%
   /* <ADD> %%= +                                                        */
   VAL(MP) = ADD;

CASEE(108)%
   /* <ADD> %%= -                                                        */
   VAL(MP) = SUBT;

CASEE(109)%
   /* <TERM> %%= <PRIMARY>                                               */
   ;

CASEE(110)%
   /* <TERM> %%= <TERM> <MULT> <PRIMARY>                                 */
   CALL EVALUATE(VAL(MPP1));

CASEE(111)%
   /* <MULT> %%= *                                                       */
   VAL(MP) = MULT;

CASEE(112)%
   /* <MULT> %%= /                                                       */
   VAL(MP) = DIV;

CASEE(113)%
   /* <MULT> %%= MOD                                                     */
   VAL(MP) = MODD ;

CASEE(114)%
   /* <PRIMARY> %%= <CONSTANT>                                           */
   CLASS(MP) = CONSTANT_CLASS;

CASEE(115)%
   /* <PRIMARY> %%= <VARIABLE>                                           */
   ;

CASEE(116)%
   /* <PRIMARY> %%= ( <EXPRESSION> )                                     */
   CALL MOVE_STACKS(MPP1,MP);

CASEE(117)%
   /* <VARIABLE> %%= <IDENTIFIER>                                        */
   CALL ACCESS_VARIABLE(0);

CASEE(118)%
   /* <VARIABLE> %%= <IDENTIFIER> ( <EXPRESSION LIST> )                  */
   EXP_CNT = FIXL(SP-1);
   CALL ACCESS_VARIABLE(CNT(SP-1));

CASEE(119)%
   /* <VARIABLE> %%= <BUILTIN FUNCTION>                                  */
   CALL EVALUATE_BUILTIN_FUNCTION(0);
```

```
CASEE(120)$
   /* <VARIABLE> $$= <BUILTIN FUNCTION> ( <EXPRESSION LIST> )        */
   EXP_CNT = FIXL(SP-1);
   CALL EVALUATE_BUILTIN_FUNCTION(CNT(SP-1));

CASEE(121)$
   /* <EXPRESSION LIST> $$= <EXPRESSION>                             */
   FIXL(MP) = EXP_CNT;
   INC_EXP_CNT;
   CNT(MP) = 1;
   CLASS_STACK(EXP_CNT) = SHL(CLASS(MP),30) OR TYPE(MP);
   IF CLASS(MP) = CONSTANT_CLASS THENN
      IF TYPE(MP) = FIXEDTYPE THENN
         EXP_STACK(EXP_CNT) = VAL(MP);
      ELSEE
         EXP_STACK(EXP_CNT) = VAR(MP);
      END;
   ELSEE
      EXP_STACK(EXP_CNT) = G1(MP);
   END;

CASEE(122)$
   /* <EXPRESSION LIST> $$= <EXPRESSION LIST> , <EXPRESSION>         */
   INC_EXP_CNT;
   CNT(MP) = CNT(MP) + 1;
   CLASS_STACK(EXP_CNT) = SHL(CLASS(SP),30) OR TYPE(SP);
   IF CLASS(SP) = CONSTANT_CLASS THENN
      IF TYPE(SP) = FIXEDTYPE THENN
         EXP_STACK(EXP_CNT) = VAL(SP);
      ELSEE
         EXP_STACK(EXP_CNT) = VAR(SP);
      END;
   ELSEE
      EXP_STACK(EXP_CNT) = G1(SP);   /* SAVE POINTER */
   END;

CASEE(123)$
   /* <CONSTANT> $$= <STRING>                                        */
   TYPE(MP) = CHRTYPE;

CASEE(124)$
   /* <CONSTANT> $$= <NUMBER>                                        */
   TYPE(MP) = FIXEDTYPE;

CASEE(125)$
   /* TERMINATION CASE */
   ;

END_CASE;

END SYN2;
```

```
   IF PROD <= 44 THENN
      CALL SYN0(PROD - 0);
   ELSEE
      IF PROD <= 86.THENN
         CALL SYN1(PROD - 45);
      ELSEE
         IF PROD <= 125 THENN
            CALL SYN2(PROD - 87);
         ELSEE
            OUTPUT = 'ILLEGAL PRODUCTION NUMBER';
         END;
      END;
   END;

END SYNTHESIS;

/*  END SEMANTICS  */
/* THE ERROR RECOVERY TABLE FOR THE GRAMMAR IS AS FOLLOWS$

      NOTE$  1)  THE COLOUMNS ARE ASSIGNED FROM RIGHT TO LEFT  AND
             2)  THE ROWS ARE ASSIGNED FROM TOP TO BOTTOM
             TO THE TERMINALS AS LISTED IN VOCABULARY.              */

DECLARE ERROR_TABLE(49) FIXED INITIAL(
   /* 0 */
   /*  98765432109876543210987654321098765432109876543210 */
   ^(1)00000000000000000000000000000000000000000000000000^,
   ^(1)00000000000000000001110000111000110000000000101110^,
   ^(1)00000000000000000001111000111101011100000000101110^,
   ^(1)00000000000000000000000000000000000000000100100000^,
   ^(1)00000000011000110000100000000000000111000110100000^,
   ^(1)00000000000011111111101100000100000101000001011101110101010 0^,
   ^(1)00000000000000000000000000000000000000000010000000^,
   ^(1)00000000000011111111101100000100000101000000000000001000010100^,
   ^(1)00000000011000110000100000000000000111000110100000^,
   ^(1)00000000000011111111101100000100000101000000111110110001010 0^,
   /* 10 */
   /*  98765432109876543210987654321098765432109876543210 */
   ^(1)00000000000000000000000000000000000001001000000000^,
   ^(1)00000000000000000000000000000000000000010000000000^,
   ^(1)00000000000000000000000000000000000100001000010100^,
   ^(1)00000000000000000000000000000000000000010000000000^,
   ^(1)00000000000000000000000000000000000000010000000000^,
   ^(1)00000000000000000000000000000000000000010000000000^,
   ^(1)00000000000000000000010000001000000000110000000000^,
   ^(1)00000000000000000000000000000000000111000100000100^,
   ^(1)00000000000000000000000000000000001000000000000000^,
   ^(1)00000000000000000000000000000000000000000000100100^,
   /* 20 */
   /*  98765432109876543210987654321098765432109876543210 */
   ^(1)00000000000000000000000000000000000000000000000100^,
   ^(1)00000000011000110000100000000000000000000110100000^,
   ^(1)00000000000000000001110000111000110000000000100100^,
   ^(1)00000000000000000001110000111000110000000000100100^,
   ^(1)00000000011000110000100000000100000000000110100000^,
   ^(1)00000000000000000000000000000100000000000010000000^,
   ^(1)00000000000000000000000000000000000000000010000000^,
   ^(1)00000000000000000000000101000000000000000000100100^,
   ^(1)00000000000000000000000000000000000000000000000100^,
   ^(1)00000000011000110000100000000000000000000110100000^,
   /* 30 */
   /*  98765432109876543210987654321098765432109876543210 */
   ^(1)00000000011000110000100000000000000000000110100000^,
   ^(1)00000000011000110000100000000000000000000110100000^,
   ^(1)00000000011000110000100000000000000000000110100100^,
   ^(1)00000000000000000000000000000000100000000000100000^,
   ^(1)00000000000000000000000000000000000000000000000100^,
   ^(1)00000000000000000000000000000000000000000010000000^,
   ^(1)000000000011000110000100000000000000000000000000000110100000^,
   ^(1)000000000011000110000100000000000000000000000000000110100000^,
   ^(1)000000000011000110000000000000000000000000000000000110100000^,
   ^(1)000000000011000110000000000000000000010000000000000110100000^,
   /* 40 */
   /*  987654321098765432109876543210987654321098765432109876543210 */
   ^(1)000000000011000110000000000000000000010000000000000110100000^,
   ^(1)000000000011000110000000000000000000000000000000000110100000^,
   ^(1)000000000011000111000000000000000000000000000000000110100000^,
   ^(1)000000000011000000000000000000000000000000000000000110100000^,
   ^(1)000000000011000000000000000000000000000000000000000110100000^,
   ^(1)000000000011000000000000000000000000000000000000000110100000^,
   ^(1)000000000011000000000000000000000000000000000000000110100000^,
   ^(1)000000000011000000000000000000000000000000000000000110100000^,
   ^(1)000000000000111111111011000001000001010000000000001100010100^,
   ^(1)000000000000111111111011000001000001010000000000001000010100^ );

/* G R O U P I N G   T A B L E */

DECLARE GROUPING_TABLE(49) FIXED INITIAL (
   0, 8, 6, 0, 0, 0, 0, 0, 1, 2, 0, 0, 6, 6, 0, 0, 0, 3, 0, 7, 0, 0, 6, 0,
   0, 0, 0, 4, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0);
```

```
 ^(1)000000000011000110000100000000000000000000000000000110100000^,
 ^(1)000000000011000110000100000000000000000000000000000110100000^,
 ^(1)000000000011000110000000000000000000000000000000000110100000^,
 ^(1)000000000011000110000000000000100000000000000000000110100000^,
 /* 40 */
 /*  987654321098765432109876543210987654321098765432109876543210 */
 ^(1)000000000011000110000000000000100000000000000000000110100000^,
 ^(1)000000000011000110000000000000000000000000000000000110100000^,
 ^(1)000000000011000111000000000000000000000000000000000110100000^,
 ^(1)000000000011000000000000000000000000000000000000000110100000^,
 ^(1)000000000011000000000000000000000000000000000000000110100000^,
 ^(1)000000000011000000000000000000000000000000000000000110100000^,
 ^(1)000000000011000000000000000000000000000000000000000110100000^,
 ^(1)000000000011000000000000000000000000000000000000000110100000^,
 ^(1)000000000000011111111101100000100000101000000000001100010100^,
 ^(1)000000000000011111111101100000100000101000000000001000010100^ );

/* G R O U P I N G   T A B L E */

DECLARE GROUPING_TABLE(49) FIXED INITIAL (
   0, 8, 6, 0, 0, 0, 0, 0, 1, 2, 0, 0, 6, 6, 0, 0, 0, 3, 0, 7, 0, 0, 6, 0,
   0, 0, 0, 4, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0);
```

```
/* A C T I O N   T A B L E */

DECLARE AT_SIZE LITERALLY '150';

DECLARE ACTION_TABLE(AT_SIZE) FIXED INITIAL (
   /* 0 */
   ^(3)0221220^,^(3)0222220^,^(3)0224141^,^(3)0403202^,^(3)0421220^,
   ^(3)0422220^,^(3)0423202^,^(3)0424202^,^(3)0425202^,^(3)0427202^,
   ^(3)0431202^,^(3)0433202^,^(3)0434202^,^(3)0440202^,^(3)0441202^,
   ^(3)0501102^,^(3)0505102^,^(3)0521120^,^(3)0522120^,^(3)0523102^,
   /* 20 */
   ^(3)0524102^,^(3)0525102^,^(3)0527102^,^(3)0531102^,^(3)0532102^,
   ^(3)0533102^,^(3)0534102^,^(3)0540102^,^(3)0541102^,^(3)0701102^,
   ^(3)0703102^,^(3)0705102^,^(3)0712102^,^(3)0713102^,^(3)0720302^,
   ^(3)0723102^,^(3)0724102^,^(3)0725102^,^(3)0727102^,^(3)0731102^,
   /* 40 */
   ^(3)0732102^,^(3)0733102^,^(3)0734102^,^(3)0740102^,^(3)0741102^,
   ^(3)1101102^,^(3)1105102^,^(3)1112102^,^(3)1113102^,^(3)1120302^,
   ^(3)1121220^,^(3)1122220^,^(3)1123102^,^(3)1124102^,^(3)1125102^,
   ^(3)1127102^,^(3)1131102^,^(3)1133102^,^(3)1134102^,^(3)1140102^,
   /* 60 */
   ^(3)1141102^,^(3)1403102^,^(3)1405104^,^(3)1420302^,^(3)1561110^,
   ^(3)1761110^,^(3)2003202^,^(3)2005202^,^(3)2012202^,^(3)2023202^,
   ^(3)2024202^,^(3)2025202^,^(3)2027202^,^(3)2031202^,^(3)2032202^,
   ^(3)2033202^,^(3)2034202^,^(3)2040202^,^(3)2041202^,^(3)2103102^,
   /* 80 */
   ^(3)2104302^,^(3)2105102^,^(3)2123102^,^(3)2124102^,^(3)2125102^,
   ^(3)2131102^,^(3)2132102^,^(3)2133102^,^(3)2134102^,^(3)2140102^,
   ^(3)2141102^,^(3)2202121^,^(3)2220121^,^(3)2301102^,^(3)2320302^,
   ^(3)2323102^,^(3)2324102^,^(3)2325102^,^(3)2327102^,^(3)2331102^,
   /* 100 */
   ^(3)2332102^,^(3)2333102^,^(3)2334102^,^(3)2340102^,^(3)2341102^,
   ^(3)3105110^,^(3)3161110^,^(3)3320302^,^(3)3325102^,^(3)3340102^,
   ^(3)3341102^,^(3)3401102^,^(3)3405102^,^(3)3420302^,^(3)3423102^,
   ^(3)3424102^,^(3)3425102^,^(3)3427102^,^(3)3431102^,^(3)3432102^,
   /* 120 */
   ^(3)3433102^,^(3)3434102^,^(3)3440102^,^(3)4020302^,^(3)4023102^,
   ^(3)4025102^,^(3)4027102^,^(3)4031102^,^(3)4032102^,^(3)4033102^,
   ^(3)4034102^,^(3)4041102^,^(3)4042102^,^(3)4043102^,^(3)6101102^,
   ^(3)6105102^,^(3)6110153^,^(3)6114111^,^(3)6115111^,^(3)6117111^,
   /* 140 */
   ^(3)6123102^,^(3)6124102^,^(3)6125102^,^(3)6127102^,^(3)6131102^,
   ^(3)6132102^,^(3)6133102^,^(3)6134102^,^(3)6140102^,^(3)6141102^,
   ^(3)6161153^ );
```

CHAPTER 5

SUMMARY AND CONCLUSIONS

A compiler writing system is presented in this thesis which is an extension of the XPL system [MK1]. The proposed system includes automatic generation of error repair tables and synthesis procedures. Also discussed is a paragraphing technique which exposes the logical structure of the program input to the compiler as well as improves its readability on the output page.

The proposed compiler system is composed of two parts. The first part is the SLR(1) language analyzer which is modified to generate, in addition to parsing tables, the error repair tables and the systhesis procedures. The second part of the compiler writing system is the XPL skeleton compler which includes the scanning routines, error repair routines, paragraphing routines, standard semantic routines, parsing routines and code emission routines for CDC 6400 computer. To produce the compiler for a language, the results from the first part of the compiler writing system are manually merged into the second part.

To generate the synthesis procedures, the language analyzer requires an augmented grammar as an input. The augmented grammar is constructed by associating the semantics (in HDG [RU1] notation) with each syntactic rule of the grammar.

For the language analyzer to generate the complete set of error repair tables, the input data has to be supplied by the compiler writer, specifying how the errors should be repaired. The error repair technique proposed in this thesis acts as a filter between the scanner and the parser. The information flowing from the scanner to the parser is intercepted, checked, and if necessary, repaired by the error repair technique as specified by the compiler writer or the language designer. The advantage of this technique is that it can be used in conjunction with the error recovery capabilities of the parser and also with any parsing algorithm.

The error repair technique, presented in this thesis, is based on symbol pair matching. It checks the syntactical validity of symbol pairs of the input program. If a pair is valid then the pair is passed to the parser, otherwise it is repaired first and then passed to the parser. It also repairs the errors caused by the absence of grouping elements, or the presence of too many grouping elements. The grouping elements are those elements or terminal symbols which occur in pairs, such as:

(	)
DO	END
PROCEDURE	END_PROC

Design consideration of a language also plays an important role in the simplicity and effectiveness of an error repair technique. As mentioned in section 3.3.2, a minor change in the grammar, i.e., that a procedure terminates with END_PROC instead of END, helped the error repair mechanism to confine the error, within the scope of a procedure, caused by missing END's for their corresponding DO's.

Similarly, the situation which leads to a locally correct but globally wrong construct, for example:

```
IF I > J THEN;
   OUTPUT = 'I IS GREATER THAN J.';
ELSE
   OUTPUT = 'I IS LESS THAN OR EQUAL TO J.';
```

could be avoided by slight modification in the grammar rather than by introducing complexity into the error repair algorithm to handle this situation. Currently, the proposed error repair technique, which is based on symbol pair matching, will not detect any error in the above-mentioned example. The ';' after the symbol 'THEN' is perfectly valid and represents, according to the grammar, an empty or null statement. To avoid as well as to repair this type of error, it is preferable and practical to modify the grammar so that the null statement is defined as 'NULL;' instead of ';'.

The proposed error repair technique certainly adds space and time overhead to the compiler, but it is considered to be negligible. The results for a fairly large grammar given in chapter 4, indicate that the total space required for the set of the generated error repair tables is 251 words. This space requirement could vary if the amount of data supplied for the 'action_table' (which specifies how the errors should be repaired) is varied. Still it is felt that the space overhead is small in view of the benifits of performing the error repair.

To evaluate the time overhead consider an input program which is syntactically correct. For each symbol scanned by the scanner, the control is transfered to the error repair algorithm which looks up an entry in 'error_table' and an entry in the grouping_table. If the scanned symbol is a grouping element or marks one of their check points, then some additional work is performed, such as, comparing counters, incrementing or decrementing counters, etc., as discussed in section 3.3.2, before control is transfered back to the calling program. Compared to the amount of work involved in the whole compilation process, the above-mentioned additional work is negligible.

On the other hand, if the input program is syntactically incorrect, then the work involved, in addition to the above-mentioned work, includes that which is required to repair the error. However, in this case, the additional amount of work pays for itself by helping the programmar to repair the syntactically incorrect input program, and consequently, reducing the number of compilations. In a pedagogical environment, where most of the students programs have syntax errors, the proposed error repair technique, despite its time overhead, will save more computer time by reducing the number of compilations required to debug the syntactically incorrect program to a correct one.

The proposed paragraphing technique is intended to expose the logical structure of the input program, as well as to improve the readability of the output listing. This paragraphing technique is not automated, due to the special cases considered to improve the readability of the output listing, such as, comments, macros, adding blank lines, etc. As these things are not part of the language it is difficult to automatically generate a paragraphing module, having the above-mentioned characterstics, for a given grammar. Instead, we have coded a paragraphing module which is sufficient for most XPL-like languages. This technique requires further investigation so that it could be automated for any given language.

APPENDIX A

SLR(1) ANALYZER, VERSION 2.0

HOW TO USE THE SLR(1) ANALYZER

The SLR(1) language analyzer is a program which accepts as input a grammar specified in BACKUS-NAUR FORM (BNF) and generates parsing tables to be used by a bottom-up parser. The capabilities of the program [DR1] have been augmented to produce tables for improved error recovery as well as to generate the synthesis procedures for the XPL skeleton compiler. The following is a description of the format of the input accepted by the modified SLR(1) analyzer.

INPUT GRAMMAR

The SLR(1) analyzer accepts input in either of two forms: the 'standard' form, which is a specification of the grammar in BNF format, or the 'augmented' form, where the semantic code is associated with each production.

STANDARD FORM:

Productions are placed one to a card. If the first column is non-blank, then the first token on the card is taken to be the left part of the production. Otherwise, the left part is

APPENDIX A

SLR(1) ANALYZER, VERSION 2.0

HOW TO USE THE SLR(1) ANALYZER

The SLR(1) language analyzer is a program which accepts as input a grammar specified in BACKUS-NAUR FORM (BNF) and generates parsing tables to be used by a bottom-up parser. The capabilities of the program [DR1] have been augmented to produce tables for improved error recovery as well as to generate the synthesis procedures for the XPL skeleton compiler. The following is a description of the format of the input accepted by the modified SLR(1) analyzer.

INPUT GRAMMAR

The SLR(1) analyzer accepts input in either of two forms: the 'standard' form, which is a specification of the grammar in BNF format, or the 'augmented' form, where the semantic code is associated with each production.

STANDARD FORM:

Productions are placed one to a card. If the first column is non-blank, then the first token on the card is taken to be the left part of the production. Otherwise, the left part is assumed to be the left part of the preceding production. The balance of the card is taken to be the right part of the production.

Any token that does not occur as a left part of any production is a terminal symbol; any token that only occurs as a left part is a GOAL SYMBOL (there should be exactly one in a grammar). All productions with the same left part must be grouped.

A token is either

(1) the character '<' followed by a blank, or

(2) any consecutive group of non-blank characters not beginning with '<', followed by a blank or the end of the card, or

(3) the character '<', followed by a non-blank character and then any string of blank and non-blank characters, up to and including the next occurance of the character '>'.

Blank cards are ignored. Cards with the character '_' in the first column are treated as comment or control cards (TOGGLES). One toggle per card is effective. The description of the toggles is given on the next page.

TOGGLE	INITIAL	DESCRIPTION WHEN TOGGLE IS ON
_I	OFF	List the input grammar card. (including text for augmented grammar)
_G	ON	List the reformated grammar. (only grammatical rules)
_C	OFF	List the configuration sets.
_F	ON	List the characterstic FSM.
_L	ON	List the look-ahead sets.
_D	ON	List the DPDA.
_P	OFF	Write the DPDA onto file OPUT3 (parsing tables).
_E	OFF	Compute and write error_table onto file OPUT2.
_T	OFF	List the decoded error_table.
_M	OFF	Compute and write grouping_table onto file OPUT2. Input data is read from file IPUT2.
_A	OFF	Compute and write action_table onto file OPUT2. Input data is read from file IPUT2.
_S	OFF	When this toggle is off, the input grammar should be in STANDARD form. When this toggle is on, the input grammar should be in AUGMENTED form.

AUGMENTED FORM:

The AUGMENTED grammar is formed by associating the semantics with each production rule of the grammar. The purpose of feeding the augmented grammar to the SLR analyzer is to automatically generate the 'synthesis' module of the compiler. To include the semantics in the grammar, it is desired to distinguish between procedure name, declaration statement, comment, production rule, statement, and end of procedure. Therefore, some control toggles are introduced to achieve this purpose. The meaning of these toggles and how to use them is explained below.

For the augmented form of a grammar, the first two columns of the cards are reserved. The second column should be blank and the first column should contain one of the following control characters (except comment or control cards):

P Starting from the third column all the blanks (if any) will be ignored and the first character string will be taken as a 'procedure name'. One card of this type must precede the first card with the control character 'R'.

D Starting from the third column, the card image will be written as it is. This control character could be used for 'declaration statements' and 'comments'. All declaration statements must fall between the P card and the R card following it.

R This control character indicates that starting from the third column is the 'production rule' which follows the specification of the standard form.

S Starting from the third column the card image will be written as it is. This control character is used if the card image contains a 'statement' of the semantic procedure. The difference between the D and the S control characters lies in their implementation. If there is no semantic code statement(s) associated with a production rule, then an empty statement (;) is automatically inserted.

E Indicates the end of a procedure. The last procedure should not be terminated by this control character.

GENERATED TABLES AND SYNTHESIS PROCEDURES

This language analyzer generates a set of parsing tables when the input grammar is in standard form. If the input grammar is in augmented form, the language analyzer, along with the parsing tables, generates a set of error repair tables and synthesis procedures, provided that the necessary data is supplied. The set of error repair tables consists of three tables, namely, 'error_table', 'action_table', and 'grouping_table'. Synthesis procedures and error repair tables are discussed below.

SYNTHESIS PROCEDURES:

The data required to generate the synthesis procedures is embedded in the augmented grammar (refer to augmented form), which is read by the analyzer as card images. The first column of the card is checked. If it is a '_', then it is considered as a comment card (refer to standard form). The character in the second column of the comment card is considered as a toggle character and its value is switched from on to off (or vice versa) and the rest of the information on the card is ignored. If the first character of the card image is the control character 'R' (which specifies production rule), then the text from column 3 to column 80, inclusive, is passed on to a procedure which analyzes the grammar. If the first column of the card image has a legal control character other than 'R', then the text from column 3 to column 80, inclusive, is passed on to the procedure which generates the synthesis procedures.

The structure of these procedures is shown in Fig. A-1, where the contents between '{' and '}' are taken from the card images (of the augmented grammar) having the control character other than 'R'. The contents between '/*' and '*/' are the production rules taken from the augmented grammar, i.e., the card images having 'R' as the control character. The rest of the code for synthesis procedure, as shown in Fig. A-1, is generated by the SLR(1) analyzer.

This structure of synthesis procedures requires that there should be, at least, one procedure within the main procedure called 'SYNTHESIS'. For this reason, it is required to have one card (in the augmented grammar) with control character 'P' before any card having a control character 'R'.

other than 'R', then the text from column 3 to column 80, inclusive, is passed on to the procedure which generates the synthesis procedures.

The structure of these procedures is shown in Fig. A-1, where the contents between '{' and '}' are taken from the card images (of the augmented grammar) having the control character other than 'R'. The contents between '/*' and '*/' are the production rules taken from the augmented grammar, i.e., the card images having 'R' as the control character. The rest of the code for synthesis procedure, as shown in Fig. A-1, is generated by the SLR(1) analyzer.

This structure of synthesis procedures requires that there should be, at least, one procedure within the main procedure called 'SYNTHESIS'. For this reason, it is required to have one card (in the augmented grammar) with control character 'P' before any card having a control character 'R'.

FIG. A-1

STRUCTURE OF THE SYNTHESIS PROCEDURE

```
/* The brace { and } delimit information that
   is supplied by the compiler writer in the
   augmented grammar.                               */



SYNTHESIS:
   PROCEDURE(PROD);
      DECLARE PROD FIXED;
      {declare statements}


{PROC1}:
   PROCEDURE(PROD_NO);
      DECLARE PROD_NO FIXED;
      {declare statements}

      DO CASE PROD_NO;

         CASEE0(0):  /* NOT USED */
            ;

         CASEE(1):
            /* production rule no.  1 */
            {statements}

         .
         .
         .

         CASEE(N):
            /* production rule no.  N */
            {statements}

      END_CASE;
   END {PROC1};
```

FIG. A-1

STRUCTURE OF THE SYNTHESIS PROCEDURE

```
/* The brace { and } delimit information that
   is supplied by the compiler writer in the
   augmented grammar.                              */



SYNTHESIS:
   PROCEDURE(PROD);
      DECLARE PROD FIXED;
      {declare statements}


{PROC1}:
   PROCEDURE(PROD_NO);
      DECLARE PROD_NO FIXED;
      {declare statements}

      DO CASE PROD_NO;

         CASEE0(0):  /* NOT USED */
            ;

         CASEE(1):
            /* production rule no.  1 */
            {statements}

         .
         .
         .

         CASEE(N):
            /* production rule no.  N */
            {statements}

      END_CASE;
   END {PROC1};
```

```
{PROC2}:
   PROCEDURE(PROD_NO);
      DECLARE PROD_NO FIXED;
      {declare statements}

      DO CASE PROD_NO;

         CASEE(N+1):
            /* production rule no.  N+1 */
            {statements}



         .
         .
         .


         CASEE(N+M):
            /* production rule no.  N+M */
            {statements}

      END_CASE;

   END {proc2};


   IF PROD <= N THENN
      CALL {PROC1}(PROD_NO);
   ELSEE
      IF PROD <= N+M THENN
         CALL {PROC2}(PROD - N + 1);
      ELSEE
         OUTPUT = 'ILLEGAL PRODUCTION NUMBER';
      END;
   END;


END SYNTHESIS;
```

```
{PROC2}:
   PROCEDURE(PROD_NO);
      DECLARE PROD_NO FIXED;
      {declare statements}

      DO CASE PROD_NO;

         CASEE(N+1):
            /* production rule no.  N+1 */
            {statements}

         .
         .
         .

         CASEE(N+M):
            /* production rule no.  N+M */
            {statements}

      END_CASE;

   END {proc2};


   IF PROD <= N THENN
      CALL {PROC1}(PROD_NO);
   ELSEE
      IF PROD <= N+M THENN
         CALL {PROC2}(PROD - N + 1);
      ELSEE
         OUTPUT = 'ILLEGAL PRODUCTION NUMBER';
      END;
   END;

END SYNTHESIS;
```

ERROR_TABLE:

The analyses of the grammar provide the data required to generate the error table. The size of this table is NXN values, where N is the number of terminal symbols in the grammar. Since the values this table is to hold are boolean values (i.e., 0 and 1), and the CDC 6400 computer's word is 60 bits long, the SLR(1) analyzer generates N bit-tables, each N bits long (N <= 60). A grammar having 60 terminal symbols is a sufficiently large grammar. The analyzer has to be modified if it has to generate an error table for a grammar having more than 60 terminal symbols.

The analyzer also generates a vocabulary of all the symbols (terminal and non-terminal) of a grammar. If two terminal symbols, say, TS1 and TS2, are given, and their corresponding locations in the vocabulary are, say, L1 and L2 respectively, (the locations are counted from zero), then to find out if, according to syntactic rules, TS1 can be followed by TS2, one has to look at word number L1, (counting from zero), of 'error_table' and its bit number L2, where bit numbers are assigned from 0 to 59 and from right to left. If this bit is on (i.e., 1), then TS1 can be followed by TS2 and this is called a legal pair. If the bit is off (i.e., 0), then TS1 cannot be followed by TS2, and this is called an illegal pair.

ERROR_TABLE:

The analyses of the grammar provide the data required to generate the error table. The size of this table is NXN values, where N is the number of terminal symbols in the grammar. Since the values this table is to hold are boolean values (i.e., 0 and 1), and the CDC 6400 computer's word is 60 bits long, the SLR(1) analyzer generates N bit-tables, each N bits long (N <= 60). A grammar having 60 terminal symbols is a sufficiently large grammar. The analyzer has to be modified if it has to generate an error table for a grammar having more than 60 terminal symbols.

The analyzer also generates a vocabulary of all the symbols (terminal and non-terminal) of a grammar. If two terminal symbols, say, TS1 and TS2, are given, and their corresponding locations in the vocabulary are, say, L1 and L2 respectively, (the locations are counted from zero), then to find out if, according to syntactic rules, TS1 can be followed by TS2, one has to look at word number L1, (counting from zero), of 'error_table' and its bit number L2, where bit numbers are assigned from 0 to 59 and from right to left. If this bit is on (i.e., 1), then TS1 can be followed by TS2 and this is called a legal pair. If the bit is off (i.e., 0), then TS1 cannot be followed by TS2, and this is called an illegal pair.

The parser which uses the parsing tables generated by the SLR(1) analyzer always has two terminal symbols at hand while it is parsing the input text. These symbols are known as 'current_symbol' and 'next_symbol'. Before this pair of symbols is used by the parsing algorithm, it is checked through the error table to determine if the pair is a legal one. If the pair at hand is legal, then it is passed on to the parsing algorithm. The question arises as to what to do if the pair is not legal, i.e., a syntactic error is encountered. This is discussed next.

ACTION_TABLE:

The action table specifies the action to be taken when a syntactic error is encountered. The table is generated by the analyzer provided the data are supplied. The details of the data format are discussed later. The size of this table is the same as the number of the data cards provided to generate this table. Each word of this table is divided into four fields. Fields one and two of this word contain the locations of the current symbol and the next symbol, respectively, in the vocabulary. The third field of this word contains the action code for the illegal pair (current_symbol, next_symbol). Depending on the action code, the fourth field of this word will contain the location of another terminal symbol in the vocabulary. This terminal symbol could be the replacement for either the current symbol or the next symbol, The parser which uses the parsing tables generated by the SLR(1) analyzer always has two terminal symbols at hand while it is parsing the input text. These symbols are known as 'current_symbol' and 'next_symbol'. Before this pair of symbols is used by the parsing algorithm, it is checked through the error table to determine if the pair is a legal one. If the pair at hand is legal, then it is passed on to the parsing algorithm. The question arises as to what to do if the pair is not legal, i.e., a syntactic error is encountered. This is discussed next.

ACTION_TABLE:

The action table specifies the action to be taken when a syntactic error is encountered. The table is generated by the analyzer provided the data are supplied. The details of the data format are discussed later. The size of this table is the same as the number of the data cards provided to generate this table. Each word of this table is divided into four fields. Fields one and two of this word contain the locations of the current symbol and the next symbol, respectively, in the vocabulary. The third field of this word contains the action code for the illegal pair (current_symbol, next_symbol). Depending on the action code, the fourth field of this word will contain the location of another terminal symbol in the vocabulary. This terminal symbol could be the replacement for either the current symbol or the next symbol, or it could be inserted between the two. The selection of these actions depends upon the action code. If the data is not provided for some illegal pair(s), then by default the action 'SKIP' is assumed, i.e., skip the next symbol.

GROUPING_TABLE:

The 'grouping_table' is generated by the analyzer, provided the data are supplied, to allow the compiler writer to issue specific error messages when a grouping error (unmatched parenthesis, DO's etc.) occurs. The details of the data format are discussed later. The size of this table is N words, where N is the number of terminal symbols in the grammar. This table is used for the grouping elements and their check points. Grouping elements are those terminal symbols which occur in pairs, such as [(,)], [DO, END], [PROCEDURE, END_PROC] etc. They must appear in the 'grouping_table' so that they can be counted. The check-points are terminal symbols which signal when to check whether the grouping elements are matched.

It is the responsibility of the language designer to write a procedure whose function is to check the grouping elements by using the 'grouping_table'. This procedure consists of one DO CASE statement, where each case statement will perform a specific function. An example of such a procedure is given in Fig. A-2, which only handles parentheses for the XPL grammar.

The 'grouping_table' contains the case number to be executed when a grouping element or any check point is encountered. As shown in Fig. A-2, when a '(' is encountered, the CASEE(1) will be executed, that is, the count of the left parentheses will be incremented by 1. It is also shown that if certain functions are yet to be implemented, a null statement ';' should be provided for that case statement. For example, when a 'DO' is encountered in the input text, the CASEE(4) is executed. Since this has a null statement, therefore, no action will be taken. CASEE0 is reserved and should not be used for grouping elements or for their check points. By default, all other terminal symbols will go to CASEE0.

```
CHECK_GROUP_ELEMENTS:
   PROCEDURE;
      DECLARE MSG CHARACTER(80);

      DO CASE GROUPING_TABLE(NEXT_SYMBOL);

         CASEE0:    /*  NOT USED  */
            ;

         CASEE(1):   /*  LEFT PARENTHESES  */
            LEFT_PAREN = LEFT_PAREN + 1;

         CASEE(2):   /*  RIGHT PARENTHESES  */
            RIGHT_PAREN = RIGHT_PAREN + 1;
            IF RIGHT_PAREN > LEFT_PAREN THENN
               OUTPUT(2) = ' ';
               MSG = '    ' !! IFORMAT(CARDCOUNT, 6) !! '     '
                         !! IFORMAT(LINE_COUNT, 6)!! '     ';
               OUTPUT(2) = MSG!!'*** ERROR:  TOO MANY'
                             !! ' RIGHT PARENTHESES.';
               RIGHT_PAREN = RIGHT_PAREN - 1;
               RP_ERROR = TRUE;
            END;

         CASEE(3):   /*  PROCEDURE  */
            ;

         CASEE(4):   /*  DO  */
            ;

         CASEE(5):   /*  END  */
            ;

         CASEE(6):   /*  ;  FIXED  CHARACTER  THEN  */
            IF LEFT_PAREN > RIGHT_PAREN THENN
               OUTPUT(2) = ' ';
               MSG = '    ' !! IFORMAT(CARDCOUNT, 6) !! '     '
                         !! IFORMAT(LINE_COUNT, 6)!! '     ';
               OUTPUT(2) = MSG!!'*** ERROR:  UNMATCHED'
                             !! ' LEFT PARENTHESES.';
               LP_ERROR = TRUE;
            ELSEE
               LEFT_PAREN = 0;
               RIGHT_PAREN = 0;
            END;
      END_CASE;

   END CHECK_GROUP_ELEMENTS;
```

FIG. A-2

FORMAT OF DATA

The data for the computation of the 'grouping_table' and the 'action_table' should be provided on file IPUT2. The data for the 'grouping_table' should be given first, followed by the data for the 'action_table'. The two sets of data should be separated by a card having EOD starting in the first column. EOD stands for 'end of data'.

GROUPING_TABLE:

Each data card for the 'grouping_table' should contain two data elements separated by at least one blank. The first data element should be a terminal symbol, say Vt. The second data element should be an integer number, say Ci. It is the user's responsibility to write a procedure which contains a DO CASE STATEMENT. Each case statement Ci of the DO CASE STATEMENT should be coded to handle the terminal symbol(s) Vt which are recognized by the scanner. Basically this code will be for incrementing some counters, for comparing counters or for making some other decisions.

The generated table will have the following form:
DECLARE GROUPING_TABLE(DIM) FIXED INITIAL(C0,C1,...., Cdim);
where DIM is the dimension (number of terminal symbols) and Ci is the case number to handle the ith terminal symbol as listed in the vocabulary. For a terminal symbol which is not

FORMAT OF DATA

The data for the computation of the 'grouping_table' and the 'action_table' should be provided on file IPUT2. The data for the 'grouping_table' should be given first, followed by the data for the 'action_table'. The two sets of data should be separated by a card having EOD starting in the first column. EOD stands for 'end of data'.

GROUPING_TABLE:

Each data card for the 'grouping_table' should contain two data elements separated by at least one blank. The first data element should be a terminal symbol, say Vt. The second data element should be an integer number, say Ci. It is the user's responsibility to write a procedure which contains a DO CASE STATEMENT. Each case statement Ci of the DO CASE STATEMENT should be coded to handle the terminal symbol(s) Vt which are recognized by the scanner. Basically this code will be for incrementing some counters, for comparing counters or for making some other decisions.

The generated table will have the following form:
DECLARE GROUPING_TABLE(DIM) FIXED INITIAL(C0,C1,...., Cdim);
where DIM is the dimension (number of terminal symbols) and Ci is the case number to handle the ith terminal symbol as listed in the vocabulary. For a terminal symbol which is not

provided on the data card, the program initializes its case number to zero. Therefore, in the user's coded procedure, case zero of the DO CASE STATEMENT should be an empty statement (;).

ACTION_TABLE:

Each data card for the action_table should contain character string data items separated by at least one blank in the following order:

CURRENT_SYMBOL NEXT_SYMBOL ACTION REPLACEMENT/INSERTION.

where CURRENT_SYMBOL is a terminal symbol.

NEXT_SYMBOL is a terminal symbol.

ACTION is one of the following:

1) REPLACE_CURRENT_SYMBOL
2) REPLACE_NEXT_SYMBOL
3) INSERT
4) SKIP

For actions 1) and 2) 'current'/'next symbol' is replaced by the terminal symbol specified by the 4th data item. For action 3) the terminal symbol specified by the 4th data item is inserted between 'current_symbol' and 'next_symbol'. Action 4) skips the 'next_symbol'. For this action the 4th data element on the data card is not required. For any illegal pair of symbols, if no other action is specified, the default action is to SKIP.

provided on the data card, the program initializes its case number to zero. Therefore, in the user's coded procedure, case zero of the DO CASE STATEMENT should be an empty statement (;).

ACTION_TABLE:

Each data card for the action_table should contain character string data items separated by at least one blank in the following order:

CURRENT_SYMBOL NEXT_SYMBOL ACTION REPLACEMENT/INSERTION.

where CURRENT_SYMBOL is a terminal symbol.
NEXT_SYMBOL is a terminal symbol.
ACTION is one of the following:

1) REPLACE_CURRENT_SYMBOL
2) REPLACE_NEXT_SYMBOL
3) INSERT
4) SKIP

For actions 1) and 2) 'current'/'next symbol' is replaced by the terminal symbol specified by the 4th data item. For action 3) the terminal symbol specified by the 4th data item is inserted between 'current_symbol' and 'next_symbol'. Action 4) skips the 'next_symbol'. For this action the 4th data element on the data card is not required. For any illegal pair of symbols, if no other action is specified, the default action is to SKIP.

The data for this table must appear in the same order as they appear in the decoded error table. Not much effort is required to produce the data, in the required order, if the following procedure is used:

First get the listing of the decoded error table. This listing has two parts. In the first part, the terminal symbols are printed out one at a time, followed by their legal successor terminal symbols. In the second part, the terminal symbols are printed out one at a time, followed by their illegal successor terminal symbols. It is only necessary to go through the second part of this listing sequentially, to write the data for the action table in the order required.

The data for this table must appear in the same order as they appear in the decoded error table. Not much effort is required to produce the data, in the required order, if the following procedure is used:

> First get the listing of the decoded error table. This listing has two parts. In the first part, the terminal symbols are printed out one at a time, followed by their legal successor terminal symbols. In the second part, the terminal symbols are printed out one at a time, followed by their illegal successor terminal symbols. It is only necessary to go through the second part of this listing sequentially, to write the data for the action table in the order required.

PROCEDURE FOR USING THE SLR(1) ANALYZER

The following are the systematic steps to use the SLR(1) analyzer.

(1) Feed the grammar to the analyzer in 'standard' form
   - If the grammar is SLR(1) then perform STEP (2).
   - If the grammar is not SLR(1) then modify the grammar and repeat STEP (1).

(2) Feed the grammar to the analyzer in 'standard' form with E and T toggles ON.

(3) With the help of the grammar build the data for 'grouping_table', and with the help of the decoded error table build the data for 'action_table'. Store the data for the grouping table and the data for the action table, separated by 'EOD', onto file IPUT2. Change the grammar from STANDARD FORM to AUGMENTED FORM and associate the semantic code with the production rules where necessary. Feed the augmented grammar to the analyzer along with the data file IPUT2 with the E, M, A, S and P toggles ON.

PROCEDURE FOR USING THE SLR(1) ANALYZER

The following are the systematic steps to use the SLR(1) analyzer.

(1) Feed the grammar to the analyzer in 'standard' form
  - If the grammar is SLR(1) then perform STEP (2).
  - If the grammar is not SLR(1) then modify the grammar and repeat STEP (1).

(2) Feed the grammar to the analyzer in 'standard' form with E and T toggles ON.

(3) With the help of the grammar build the data for 'grouping_table', and with the help of the decoded error table build the data for 'action_table'. Store the data for the grouping table and the data for the action table, separated by 'EOD', onto file IPUT2. Change the grammar from STANDARD FORM to AUGMENTED FORM and associate the semantic code with the production rules where necessary. Feed the augmented grammar to the analyzer along with the data file IPUT2 with the E, M, A, S and P toggles ON.

APPENDIX B

TEST PROGRAMS

This Appendix contains the test programs submitted to a compiler generated by the proposed compiler writing system. Each test program is written to highlight the capabilities of the error repair and the paragraphing of this study. Following is a brief summary of each test program followed by the input test programs and their paragraphed output.

Test Program No. 1: A syntactically correct program highlighting the paragraphing of a 'declare' statement and operator precedence.

Test Program No. 2: A syntactically incorrect program highlighting the error repair performed on missing parenthesis, missing statement terminator, missing operator, invalid punctuation, more than one consecutive arithmetic operator and incorrect use of keyword 'CALL' after '=' sign. The paragraphed output is of the repaired input program highlighting the paragraphing of 'declare' and 'procedure' statements.

Test Program No. 3: A syntactically correct program highlighting the paragraphing of 'declare' statements

APPENDIX B

TEST PROGRAMS

This Appendix contains the test programs submitted to a compiler generated by the proposed compiler writing system. Each test program is written to highlight the capabilities of the error repair and the paragraphing of this study. Following is a brief summary of each test program followed by the input test programs and their paragraphed output.

Test Program No. 1: A syntactically correct program highlighting the paragraphing of a 'declare' statement and operator precedence.

Test Program No. 2: A syntactically incorrect program highlighting the error repair performed on missing parenthesis, missing statement terminator, missing operator, invalid punctuation, more than one consecutive arithmetic operator and incorrect use of keyword 'CALL' after '=' sign. The paragraphed output is of the repaired input program highlighting the paragraphing of 'declare' and 'procedure' statements.

Test Program No. 3: A syntactically correct program highlighting the paragraphing of 'declare' statements declaring an array with a list of initial values; paragraphing of nested 'do' statements, 'if-then-else' statements and comments.

Test Program No. 4: A syntactically correct program highlighting the paragraphing of 'declare' statements declaring a variable with a initial value; paragraphing of 'procedures' and adding of blank lines between two consecutive 'procedures', and also the truncation of the source statement which could not fit on an output line.

Test Program No. 5: A syntactically correct program which is exactly the same as test program no. 4, except at macros are used representing the text which would otherwise require paragraphing. The comparision of the paragraphed output of test program no. 4 and no. 5 will highlight the difference in paragraphing. This program also highlights the fact that the first symbol in the macro text plays a role in paragraphing.

Test Program No. 6: A syntactically correct program designed to highlight the weakness of the paragraphing when macros are used representing text which would otherwise require paragraphing. In this test program the macro 'END_CASE' is declared to represent the text 'END;END'. The first 'END' of the macro text will play a role in the paragraphing and will cause the indentation level to be decremented one level to the left and the macro name will be appended in the output buffer;

therefore, deforming the structure of the 'DO CASE' statement. However, the indentation level will be adjusted properly for the next output line.

Test Program No. 7: A syntactically incorrect program obtained from test program no. 5 by introducing several syntax errors including some in the macro text. This program highlights the error repair performed on missing parentheses, missing statement terminators, unmatched DO's which are confined within the scope of a procedure. The error repair performed on the syntactically incorrect macro text, and its effect on the paragraphed output are also illustrated. A macro 'ELSEE', instead of being declared to represent the text 'END;ELSE DO;', is declared to represent the text 'END ELSE DO '. Error_repair is performed on the macro text by providing a ';' after 'END' and a ';' after 'DO'. But the first ';' will fall within the macro text while the second ';' will fall outside the macro text. Therefore, the second ';' is printed out following the macro name 'elsee'.

TEST PROGRAM NO. 1

```
/* ]$Z ]$N ]$D ]$E ]$B ]$H */

DECLARE (I,J,L,M) FIXED;
I = 5;
J = 2 * I;
L = 5 *(I + J) * J * I + J;
M = 0;
DO WHILE L NE 0;
   M = M + 1;
   L = L - 1;
END;
OUTPUT(1) = M;
IF M > 3000 THEN
   OUTPUT = 'RESULT IS > 3000';
ELSE
   OUTPUT = 'RESULT IS <= 3000';
EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF
```

CONCORDIA UNIVERSITY - X P L COMPILER - VERSION 5.0

```
   /* $Z $M $D $E $B $H */

   DECLARE (I,
            J,
            L,
            M) FIXED;
   I = 5;
   J = 2*I;
   L = 5*(I + J)*J*I + J;
   M = 0;
   DO WHILE L NE 0;
      M = M + 1;
      L = L - 1;
   END;
   OUTPUT(1) = M;
   IF M > 3000 THEN
      OUTPUT = 'RESULT IS > 3000';
   ELSE
      OUTPUT = 'RESULT IS <= 3000';
 EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF
```

```
17 CARDS WERE CHECKED.
NO ERRORS WERE DETECTED IN SCAN PHASE.
NO ERRORS WERE DETECTED IN PARSE PHASE.
MAXIMUM SYMBOL TABLE ENTRIES ARE 6  (LIMIT IS 500)
MAXIMUM MACRO TABLE ENTRIES ARE 0  (LIMIT IS 511)
MAXIMUM STRING TABLE ENTRIES ARE 59  (LIMIT IS 700)
MAXIMUM STRING TABLE LENGTH IS 400  (LIMIT IS 6500)
NUMBER OF STRING ENTRIES IS 91 NUMBER OF COLLISIONS IS 48
COMPILER IMPLEMENTATION RESTRICTS CONSTANT STRINGS TO 256 CHARACTERS.
MAXIMUM GRAPH STACK SIZE IS 81  (LIMIT IS 1600)
WORDS OF CODE GENERATED = 163B
TOTAL STORAGE NEEDED INCLUDING MONITOR IS 3227B WORDS
NUMBER OF BLOCKS IN OBJECT FILE IS 1
NUMBER OF BUFFER SWAP REQUESTS WAS 1
NUMBER OF CODE BUFFERS INPUT/OUTPUT = 1
```

```
************ SUMMARY OF ERRORS REPAIRED. ************

CARD NO.   LINE NO.     ERROR MESSAGES AND CORRECTIONS.
________   ________     ______________________________
```

TEST PROGRAM NO. 2

```
/* ]$% */
DECLARE(A,B,C,D)))FIXED$
DECLARE(E,F)FIXED
G CHARACTER 10)

PRINT;
PROCEDURE ,
G = 'ABCD'
OUTPUT 1) = G
E=5 5
F = 10 (A
END_PROC PRINT;

A=10
B=A +*/ A
C=(( A * ( B + A )/( A
D=B MOD A
OUTPUT = CALL D

CALL PRINT;
EOF EOF
```

```
CONCORDIA UNIVERSITY  -  X P L COMPILER --  VERSION 5.0

  1 |                                                                   |
  2 |                                                                   |
  3 |  /* ]$Z */                                                        |
  4 | DECLARE (A,                                                       |
  5 |          B,                                                       |
  6 |          C,                                                       |
  7 |          D) FIXED;                                                |
  8 | DECLARE (E,                                                       |
  9 |          F) FIXED,                                                |
 10 |          G CHARACTER(10);                                         |
 11 |                                                                   |
 12 |                                                                   |
 13 |                                                                   |
 14 | PRINT:                                                            |
 15 |    PROCEDURE;                                                     |
 16 |       G = 'ABCD';                                                 | PRINT
 17 |       OUTPUT(1) = G;                                              | PRINT
 18 |       E = 5 + 5;                                                  | PRINT
 19 |       F = 10 + (A);                                               | PRINT
 20 |    END_PROC PRINT;                                                |
 21 |                                                                   |
 22 | A = 10;                                                           |
 23 | B = A + A;                                                        |
 24 | C = ((A*(B + A)/(A)));                                            |
 25 | D = B MOD A;                                                      |
 26 | OUTPUT = D;                                                       |
 27 |                                                                   |
 28 | CALL PRINT;                                                       |
 29 |EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF |
21 CARDS WERE CHECKED.
27 ERRORS WERE DETECTED AND REPAIRED IN SCAN PHASE.
NO ERRORS WERE DETECTED IN PARSE PHASE.
MAXIMUM SYMBOL TABLE ENTRIES ARE 10  (LIMIT IS 500)
MAXIMUM MACRO TABLE ENTRIES ARE 0  (LIMIT IS 511)
MAXIMUM STRING TABLE ENTRIES ARE 62  (LIMIT IS 700)
MAXIMUM STRING TABLE LENGTH IS 379  (LIMIT IS 6500)
NUMBER OF STRING ENTRIES IS 97 NUMBER OF COLLISIONS IS 52
COMPILER IMPLEMENTATION RESTRICTS CONSTANT STRINGS TO 256 CHARACTERS.
MAXIMUM GRAPH STACK SIZE IS 45  (LIMIT IS 1600)
WORDS OF CODE GENERATED = 161B
TOTAL STORAGE NEEDED INCLUDING MONITOR IS 3225B WORDS
NUMBER OF BLOCKS IN OBJECT FILE IS 1
NUMBER OF BUFFER SWAP REQUESTS WAS 1
NUMBER OF CODE BUFFERS INPUT/OUTPUT = 1


        2    A                            FIXED     SCALAR          AT 3151(0)
                                            18, 17, 16, 16, 16, 15, 15, 12
        2    B                            FIXED     SCALAR          AT 3152(0)
                                            17, 16, 16
        2    C                            FIXED     SCALAR          AT 3153(0)
                                            17
        2    D                            FIXED     SCALAR          AT 3154(0)
                                            20, 18
        4    E                            FIXED     SCALAR          AT 3155(0)
                                            11
        4    F                            FIXED     SCALAR          AT 3156(0)
                                            12
        6    G                            CHARACTER SCALAR          AT 3157(0)
                                            10, 9
        8    PRINT                        FIXED     PROCEDURE       AT 3164(0)
                                            20
```

```
*************  SUMMARY OF ERRORS REPAIRED.  ************

CARD NO.   LINE NO.     ERROR MESSAGES AND CORRECTIONS.
________   ________     _______________________________

    2          7        *** ERROR$  TOO MANY RIGHT PARENTHESES.
                        *** )   IS SKIPPED.

    2          7        *** ERROR$  TOO MANY RIGHT PARENTHESES.
                        *** )   IS SKIPPED.

    2          7        *** ILLEGAL SYMBOL PAIR $   FIXED   $ .
                        *** $   IS REPLACED BY   ; .

    4          9        *** ILLEGAL SYMBOL PAIR $   FIXED   <IDENTIFIER> .
                        *** CORRECTED SYMBOL SEQUENCE IS $   FIXED   ,   <IDENTIFIER> .

    4         10        *** ILLEGAL SYMBOL PAIR $   CHARACTER   <NUMBER> .
                        *** CORRECTED SYMBOL SEQUENCE IS $   CHARACTER   (   <NUMBER> .

    6         10        *** ILLEGAL SYMBOL PAIR $   )   <IDENTIFIER> .
                        *** CORRECTED SYMBOL SEQUENCE IS $   )   ;   <IDENTIFIER> .

    7         12        *** ILLEGAL SYMBOL PAIR $   ;   PROCEDURE .
                        *** ;   IS REPLACED BY   $ .

    7         15        *** ILLEGAL SYMBOL PAIR $   PROCEDURE   , .
                        *** ,   IS REPLACED BY   ; .

    9         16        *** ILLEGAL SYMBOL PAIR $   <STRING>   OUTPUT .
                        *** CORRECTED SYMBOL SEQUENCE IS $   <STRING>   ,   OUTPUT .

    9         17        *** ILLEGAL SYMBOL PAIR $   OUTPUT   <NUMBER> .
                        *** CORRECTED SYMBOL SEQUENCE IS $   OUTPUT   (   <NUMBER> .

   10         17        *** ILLEGAL SYMBOL PAIR $   <IDENTIFIER>   <IDENTIFIER> .
                        *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   ;   <IDENTIFIER> .

   10         18        *** ILLEGAL SYMBOL PAIR $   <NUMBER>   <NUMBER> .
                        *** CORRECTED SYMBOL SEQUENCE IS $   <NUMBER>   +   <NUMBER> .

   11         18        *** ILLEGAL SYMBOL PAIR $   <NUMBER>   <IDENTIFIER> .
                        *** CORRECTED SYMBOL SEQUENCE IS $   <NUMBER>   ;   <IDENTIFIER> .

   11         19        *** ILLEGAL SYMBOL PAIR $   <NUMBER>   ( .
                        *** CORRECTED SYMBOL SEQUENCE IS $   <NUMBER>   +   ( .

   12         19        *** ILLEGAL SYMBOL PAIR $   <IDENTIFIER>   END_PROC .

   12         19        *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                        *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   )   END_PROC .
```

```
12     19     *** ILLEGAL SYMBOL PAIR $,  )   END_PROC .
              *** CORRECTED SYMBOL SEQUENCE IS $   )   ;   END_PROC .

15     22     *** ILLEGAL SYMBOL PAIR $.  <NUMBER>   <IDENTIFIER> .
              *** CORRECTED SYMBOL SEQUENCE IS $   <NUMBER>   ;   <IDENTIFIER> .

15     23     *** ILLEGAL SYMBOL PAIR $   +   * .
              *** *   IS SKIPPED.

15     23     *** ILLEGAL SYMBOL PAIR $   +   / .
              *** /   IS SKIPPED.

16     23     *** ILLEGAL SYMBOL PAIR $   <IDENTIFIER>   <IDENTIFIER> .
              *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   ;   <IDENTIFIER> .

17     24     *** ILLEGAL SYMBOL PAIR $   <IDENTIFIER>   <IDENTIFIER> .

17     24     *** ERROR$  UNMATCHED LEFT PARENTHESIS.
              *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   )   <IDENTIFIER> .

17     24     *** ILLEGAL SYMBOL PAIR $   )   <IDENTIFIER> .

17     24     *** ERROR$  UNMATCHED LEFT PARENTHESIS.
              *** CORRECTED SYMBOL SEQUENCE IS $   )   )   <IDENTIFIER> .

17     24     *** ILLEGAL SYMBOL PAIR $   )   <IDENTIFIER> .

17     24     *** ERROR$  UNMATCHED LEFT PARENTHESIS.
              *** CORRECTED SYMBOL SEQUENCE IS $   )   )   <IDENTIFIER> .

17     24     *** ILLEGAL SYMBOL PAIR $   )   <IDENTIFIER> .
              *** CORRECTED SYMBOL SEQUENCE IS $   )   ;   <IDENTIFIER> .

18     25     *** ILLEGAL SYMBOL PAIR $   <IDENTIFIER>   OUTPUT .
              *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   ;   OUTPUT .

18     26     *** ILLEGAL SYMBOL PAIR $   =   CALL .
              *** CALL   IS SKIPPED.

20     26     *** ILLEGAL SYMBOL PAIR $   <IDENTIFIER>   CALL .
              *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   ;   CALL .
```

TEST PROGRAM NO. 3

```
     /* $L */
DECLARE I    FIXED;
       DECLARE     C     CHARACTER    (80  ) ;
  DECLARE   J  (  10)  FIXED  INITIAL (0,1,2,  3, 4, 5,6,7,8,9,10);
DECLARE (K  ,L,    M  )  FIXED;
     /* COMMENT IN BEGINING */  C    = ' TEST IS SUCCESSFUL.'   ;
DO   I=OTO 10;
        IF J - ( I)    =K     THEN    /* TEST */  /* TEST THE COMMENT */
OUTPUT=     'K = '     ||    J(I);
   ELSE
   OUTPUT  =    /* COMMENT IN MIDDLE */    'K IS NOT FOUND YET.'    ;
    IF J(L)      =     L THEN    OUTPUT='L = '     ||  J(I);   ELSE
  DO; IF J(I)>L  THEN OUTPUT='L IS NOT IN THE LIST J.';
  ELSE /* TEST COMMENT */    DO;OUTPUT='L COULD BE IN THE LIST J.';
                OUTPUT    = 'KEEP  TRYING .';
END;  END;
IF J(I)   = M THEN/* JUST A COMMENT*/  OUTPUT   =  'M = '    ||  J(I)   ;
 M=K/L-M+10/M    *L     +    K   ; OUTPUT    =  M;

 OUTPUT  = C;     END  ;     /*    TEST   THE  COMMENT   */
EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF
```

CONCORDIA UNIVERSITY - X P L COMPILER - VERSION 5.0

```


   /* $L */
DECLARE I FIXED;
DECLARE C CHARACTER(80);
DECLARE J(10) FIXED INITIAL(
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
DECLARE (K,
         L,
         M) FIXED;

   /* COMMENT IN BEGINING */
C = ' TEST IS SUCCESSFUL.';
DO I = 0 TO 10;
   IF J(I) = K THEN   /* TEST */  /* TEST THE COMMENT */
      OUTPUT = 'K = ' || J(I);
   ELSE
      OUTPUT =    /* COMMENT IN MIDDLE */'K IS NOT FOUND YET.';
   IF J(L) = L THEN
      OUTPUT = 'L = ' || J(I);
   ELSE
      DO;
         IF J(I) > L THEN
            OUTPUT = 'L IS NOT IN THE LIST J.';
         ELSE /* TEST COMMENT */
            DO;
               OUTPUT = 'L COULD BE IN THE LIST J.';
               OUTPUT = 'KEEP  TRYING .';
            END;
      END;
   IF J(I) = M THEN /* JUST A COMMENT*/
      OUTPUT = 'M = ' || J(I);
   M = K/L - M + 10/M*L + K;
   OUTPUT = M;

   OUTPUT = C;
END;     /*    TEST   THE  COMMENT  */
EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF
```

```
21 CARDS WERE CHECKED.
NO ERRORS WERE DETECTED IN SCAN PHASE.
NO ERRORS WERE DETECTED IN PARSE PHASE.
MAXIMUM SYMBOL TABLE ENTRIES ARE 8  (LIMIT IS 500)
MAXIMUM MACRO TABLE ENTRIES ARE 0  (LIMIT IS 511)
MAXIMUM STRING TABLE ENTRIES ARE 67  (LIMIT IS 700)
MAXIMUM STRING TABLE LENGTH IS 482  (LIMIT IS 6500)
NUMBER OF STRING ENTRIES IS 135 NUMBER OF COLLISIONS IS 91
COMPILER IMPLEMENTATION RESTRICTS CONSTANT STRINGS TO 256 CHARACTERS.
MAXIMUM GRAPH STACK SIZE IS 162  (LIMIT IS 1600)
WORDS OF CODE GENERATED = 336B
TOTAL STORAGE NEEDED INCLUDING MONITOR IS 3402B WORDS
NUMBER OF BLOCKS IN OBJECT FILE IS 2
NUMBER OF BUFFER SWAP REQUESTS WAS 7
NUMBER OF CODE BUFFERS INPUT/OUTPUT = 2



          2    I                  FIXED     SCALAR                  AT 3151(0)
                                     20, 17, 17, 13, 12, 9, 8
          3    C                  CHARACTER SCALAR                  AT 3152(0)
                                     20, 6
          4    J                  FIXED     VECTOR  (  10 )         AT 3163(0)
                                     17, 17, 13, 12, 12, 9, 8
          5    K                  FIXED     SCALAR                  AT 3176(0)
                                     18, 18, 8
          5    L                  FIXED     SCALAR                  AT 3177(0)
                                     18, 18, 13, 12, 12
          5    M                  FIXED     SCALAR                  AT 3200(0)
                                     18, 18, 18, 18, 17
```

```
*************  SUMMARY OF ERRORS REPAIRED.  *************

CARD NO.   LINE NO.     ERROR MESSAGES AND CORRECTIONS.
--------   --------     -------------------------------
```

TEST PROGRAM NO. 4

```
/*    $L     */

/*****************************/
/*   FORMAT PROCEDURES      */
/*****************************/

DECLARE   STRINGLIMIT    LITERALLY  '160',
         X80            CHARACTER(80) INITIAL('                    '),
          CARDCOUNT      FIXED   INITIAL(0),
          BUFFER         CHARACTER(80),
          CONTROL(64)    FIXED;


PAD$    /*  TO PAD THE STRING */ PROCEDURE(STRING,WIDTH)CHARACTER(
STRINGLIMIT);DECLARE (NUMBER,WIDTH,L)FIXED,STRING CHARACTER
(STRINGLIMIT);L=LENGTH(STRING);IF L<WIDTH THEN DO;STRING
=SUBSTR(X80,0,WIDTH-L)||STRING;END;RETURN STRING; /* IT IS DONE */
END_PROC PAD;IFORMAT$ /* TO PAD NUMBERS */ PROCEDURE(NUMBER,WIDTH)
CHARACTER(STRINGLIMIT);DECLARE (NUMBER,WIDTH,L)FIXED,
STRING CHARACTER(STRINGLIMIT);

STRING = NUMBER;  /* CHANGE NUMBER TO STRING */
L=LENGTH(STRING);IF L<WIDTH THEN DO;STRING=SUBSTR(X80,0,WIDTH-L)
||STRING;END;RETURN STRING; /* IT IS DONE */  END_PROC IFORMAT;
FLUSH_CARDS$  PROCEDURE;  /*  FLUSHES THE CARDS */
OUTPUT='***** FLUSH PROGRAM *****';IF CONTROL(BYTE('L')) OR
CONTROL(BYTE('M'))THEN DO;DO WHILE LENGTH(BUFFER)NE 0;BUFFER=INPUT;
CARDCOUNT=CARDCOUNT+1;IF LENGTH(BUFFER)>0THEN DO;IF
CONTROL(BYTE('M'))THEN DO;OUTPUT=IFORMAT(CARDCOUNT,7)||' '||BUFFER;
END;ELSE DO;IF CONTROL(BYTE('L'))THEN DO;
OUTPUT=IFORMAT(CARDCOUNT,7)||' '||BUFFER||'|';END;END;END;END;END;
END_PROC FLUSH_CARDS;

BUFFER=IFORMAT(10,5);BUFFER=BUFFER||PAD('     IS TEN',10);
CALL FLUSH_CARDS;EOF EOF EOF EOF
```

```



   /*   $L      */

/*****************************/
/*   FORMAT PROCEDURES       */
/*****************************/

  DECLARE STRINGLIMIT LITERALLY '160',
          X80 CHARACTER(80) INITIAL('                                        '),
          CARDCOUNT FIXED INITIAL(0),
          BUFFER CHARACTER(80),
          CONTROL(64) FIXED;



  PAD:    /*  TO PAD THE STRING */
     PROCEDURE(STRING, WIDTH) CHARACTER( STRINGLIMIT);
        DECLARE (NUMBER,                                                   PAD
                 WIDTH,                                                    PAD
                 L) FIXED,                                                 PAD
                 STRING CHARACTER( STRINGLIMIT);                           PAD
        L = LENGTH(STRING);                                                PAD
        IF L < WIDTH THEN                                                  PAD
           DO;                                                             PAD
              STRING = SUBSTR(X80, 0, WIDTH - L) || STRING;                PAD
           END;                                                            PAD
        RETURN STRING; /* IT IS DONE */                                    PAD
     END_PROC PAD;


  IFORMAT: /* TO PAD NUMBERS */
     PROCEDURE(NUMBER, WIDTH) CHARACTER( STRINGLIMIT);
        DECLARE (NUMBER,                                                   IFORMAT
                 WIDTH,                                                    IFORMAT
                 L) FIXED,                                                 IFORMAT
                 STRING CHARACTER( STRINGLIMIT);                           IFORMAT
                                                                           IFORMAT
        STRING = NUMBER;  /* CHANGE NUMBER TO STRING */                    IFORMAT
        L = LENGTH(STRING);                                                IFORMAT
        IF L < WIDTH THEN                                                  IFORMAT
           DO;                                                             IFORMAT
              STRING = SUBSTR(X80, 0, WIDTH - L) || STRING;                IFORMAT
           END;                                                            IFORMAT
        RETURN STRING; /* IT IS DONE */                                    IFORMAT
     END_PROC IFORMAT;
```

```
!                                                                           !
!                                                                           !
!  FLUSH_CARDS:                                                             !
!     PROCEDURE;  /* FLUSHES THE CARDS */                                   !
!        OUTPUT = '***** FLUSH PROGRAM *****';                              ! FLUSH_CARDS
!        IF CONTROL(BYTE('L')) OR CONTROL(BYTE('M')) THEN                   ! FLUSH_CARDS
!           DO;                                                             ! FLUSH_CARDS
!              DO WHILE LENGTH(BUFFER) NE 0;                                ! FLUSH_CARDS
!                 BUFFER = INPUT;                                           ! FLUSH_CARDS
!                 CARDCOUNT = CARDCOUNT + 1;                                ! FLUSH_CARDS
!                 IF LENGTH(BUFFER) > 0 THEN                                ! FLUSH_CARDS
!                    DO;                                                    ! FLUSH_CARDS
!                       IF CONTROL(BYTE('M')) THEN                          ! FLUSH_CARDS
!                          DO;                                              ! FLUSH_CARDS
!                             OUTPUT = IFORMAT(CARDCOUNT, 7) || ' |' || BUFFER; ! FLUSH_CARDS
!                          END;                                             ! FLUSH_CARDS
!                       ELSE                                                ! FLUSH_CARDS
!                          DO;                                              ! FLUSH_CARDS
!                             IF CONTROL(BYTE('L')) THEN                    ! FLUSH_CARDS
!                                DO;                                        ! FLUSH_CARDS
!                                   OUTPUT = IFORMAT(CARDCOUNT, 7) || ' |' || ! FLUSH_CARDS
!                                      BUFFER || '|';                       ! FLUSH_CARDS
!                                END;                                       ! FLUSH_CARDS
!                          END;                                             ! FLUSH_CARDS
!                    END;                                                   ! FLUSH_CARDS
!              END;                                                         ! FLUSH_CARDS
!           END;                                                            ! FLUSH_CARDS
!     END_PROC FLUSH_CARDS;                                                 !
!                                                                           !
!  BUFFER = IFORMAT(10, 5);                                                 !
!  BUFFER = BUFFER || PAD('     IS TEN', 10);                               !
!  CALL FLUSH_CARDS;                                                        !
!EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF !
```

```
37 CARDS WERE CHECKED.
NO ERRORS WERE DETECTED IN SCAN PHASE.
NO ERRORS WERE DETECTED IN PARSE PHASE.
MAXIMUM SYMBOL TABLE ENTRIES ARE 14  (LIMIT IS 500)
MAXIMUM MACRO TABLE ENTRIES ARE 2  (LIMIT IS 511)
MAXIMUM STRING TABLE ENTRIES ARE 67  (LIMIT IS 700)
MAXIMUM STRING TABLE LENGTH IS 453  (LIMIT IS 6500)
NUMBER OF STRING ENTRIES IS 208 NUMBER OF COLLISIONS IS 163
COMPILER IMPLEMENTATION RESTRICTS CONSTANT STRINGS TO 256 CHARACTERS.
MAXIMUM GRAPH STACK SIZE IS 98  (LIMIT IS 1600)
WORDS OF CODE GENERATED = 512B
TOTAL STORAGE NEEDED INCLUDING MONITOR IS 3556B WORDS
NUMBER OF BLOCKS IN OBJECT FILE IS 2
NUMBER OF BUFFER SWAP REQUESTS WAS 7
NUMBER OF CODE BUFFERS INPUT/OUTPUT = 2


      9    X80                       CHARACTER SCALAR            AT 3151(0)
                                        25, 19
     10    CARDCOUNT                 FIXED     SCALAR            AT 3162(0)
                                        33, 31, 30, 30
     11    BUFFER                    CHARACTER SCALAR            AT 3163(0)
                                        36, 36, 36, 33, 31, 30, 29, 29
     12    CONTROL                   FIXED     VECTOR  (  64 )   AT 3174(0)
                                        32, 31, 29, 28
     17    PAD                       CHARACTER PROCEDURE         AT 3323(0)
                                        36
     18       PARAMETER              CHARACTER SCALAR            AT 3300(0)
     17       PARAMETER              FIXED     SCALAR            AT 3276(0)
     21    IFORMAT                   CHARACTER PROCEDURE         AT 3375(0)
                                        30, 33, 31
     21       PARAMETER              FIXED     SCALAR            AT 3347(0)
     21       PARAMETER              FIXED     SCALAR            AT 3350(0)
     28    FLUSH_CARDS               FIXED     PROCEDURE         AT 3442(0)
                                        37
```

```
MACRO DEFINITIONS:

NAME                    TEXT

STRINGLIMIT             160
                          22, 21, 18, 17
```

```
************  SUMMARY OF ERRORS REPAIRED.  ************

CARD NO.   LINE NO.    ERROR MESSAGES AND CORRECTIONS.
________   ________    _______________________________
```

TEST PROGRAM NO. 5

```
   /*    $L     */
/*******************************/
/*   FORMAT PROCEDURES       */
/*******************************/

DECLARE   STRINGLIMIT    LITERALLY  '160',
 THENN  LITERALLY 'THEN DO;' ,
 ELSEE LITERALLY 'END;ELSE DO;' ,
        X80          CHARACTER(80) INITIAL('                                    '),
         CARDCOUNT       FIXED   INITIAL(0),
         BUFFER          CHARACTER(80),
         CONTROL(64)     FIXED;


PAD$   /*  TO PAD THE STRING */ PROCEDURE(STRING,WIDTH)CHARACTER(
STRINGLIMIT);DECLARE (NUMBER,WIDTH,L)FIXED,STRING CHARACTER
(STRINGLIMIT);L=LENGTH(STRING);IF L<WIDTH THENN STRING
=SUBSTR(X80,0,WIDTH-L)||STRING;END;RETURN STRING; /* IT IS DONE */
END_PROC PAD;IFORMAT$ /* TO PAD NUMBERS */ PROCEDURE(NUMBER,WIDTH)
CHARACTER(STRINGLIMIT);DECLARE (NUMBER,WIDTH,L)FIXED,
STRING CHARACTER(STRINGLIMIT);

STRING = NUMBER;  /* CHANGE NUMBER TO STRING */
L=LENGTH(STRING);IF L<WIDTH THENN STRING=SUBSTR(X80,0,WIDTH-L)
||STRING;END;RETURN STRING; /* IT IS DONE */  END_PROC IFORMAT;
FLUSH_CARDS$  PROCEDURE;  /*  FLUSHES THE CARDS *7
OUTPUT='***** FLUSH PROGRAM *****';IF CONTROL(BYTE('L')) OR
CONTROL(BYTE('M'))THENN DO WHILE LENGTH(BUFFER)NE 0;BUFFER=INPUT;
CARDCOUNT=CARDCOUNT+1;IF LENGTH(BUFFER)>0THENN IF
CONTROL(BYTE('M'))THENN OUTPUT=IFORMAT(CARDCOUNT,7)||' |'||BUFFER;
ELSEE IF CONTROL(BYTE('L'))THENN
OUTPUT=IFORMAT(CARDCOUNT,7)||' |'||BUFFER||'|';END;END;END;END;END;
END_PROC FLUSH_CARDS;

BUFFER=IFORMAT(10,5);BUFFER=BUFFER||PAD('     IS TEN',10);
CALL FLUSH_CARDS;EOF EOF EOF EOF
```

```



    /*    $L     */

  /*****************************/
  /*   FORMAT PROCEDURES      */
  /*****************************/

   DECLARE STRINGLIMIT LITERALLY '160',
           THENN LITERALLY 'THEN DO;',
           ELSEE LITERALLY 'END;ELSE DO;',
           X80 CHARACTER(80) INITIAL('                                  '),
           CARDCOUNT FIXED INITIAL(0),
           BUFFER CHARACTER(80),
           CONTROL(64) FIXED;



   PAD:    /*  TO PAD THE STRING */
      PROCEDURE(STRING, WIDTH) CHARACTER( STRINGLIMIT);
         DECLARE (NUMBER,                                          PAD
                  WIDTH,                                           PAD
                  L) FIXED,                                        PAD
                  STRING CHARACTER( STRINGLIMIT);                  PAD
         L = LENGTH(STRING);                                       PAD
         IF L < WIDTH THENN                                        PAD
            STRING = SUBSTR(X80, 0, WIDTH - L) || STRING;          PAD
         END;                                                      PAD
         RETURN STRING; /* IT IS DONE */                           PAD
      END_PROC PAD;


   IFORMAT: /* TO PAD NUMBERS */
      PROCEDURE(NUMBER, WIDTH) CHARACTER( STRINGLIMIT);
         DECLARE (NUMBER,                                          IFORMAT
                  WIDTH,                                           IFORMAT
                  L) FIXED,                                        IFORMAT
                  STRING CHARACTER( STRINGLIMIT);                  IFORMAT
                                                                   IFORMAT
         STRING = NUMBER;  /* CHANGE NUMBER TO STRING */           IFORMAT
         L = LENGTH(STRING);                                       IFORMAT
         IF L < WIDTH THENN                                        IFORMAT
            STRING = SUBSTR(X80, 0, WIDTH - L) || STRING;          IFORMAT
         END;                                                      IFORMAT
         RETURN STRING; /* IT IS DONE */                           IFORMAT
      END_PROC IFORMAT;
```

```
48 !                                                                        !
49 !                                                                        !
50 !   FLUSH_CARDS:                                                         !
51 !     PROCEDURE;  /*  FLUSHES THE CARDS */                               !
52 !       OUTPUT = '***** FLUSH PROGRAM *****';                            ! FLUSH_CARDS
53 !       IF CONTROL(BYTE('L')) OR CONTROL(BYTE('M')) THENN                ! FLUSH_CARDS
54 !         DO WHILE LENGTH(BUFFER) NE 0;                                  ! FLUSH_CARDS
55 !           BUFFER = INPUT;                                              ! FLUSH_CARDS
56 !           CARDCOUNT = CARDCOUNT + 1;                                   ! FLUSH_CARDS
57 !           IF LENGTH(BUFFER) > 0 THENN                                  ! FLUSH_CARDS
58 !             IF CONTROL(BYTE('M')) THENN                                ! FLUSH_CARDS
59 !               OUTPUT = IFORMAT(CARDCOUNT, 7) || ' |' || BUFFER;        ! FLUSH_CARDS
60 !             ELSEE                                                      ! FLUSH_CARDS
61 !               IF CONTROL(BYTE('L')) THENN                              ! FLUSH_CARDS
62 !                 OUTPUT = IFORMAT(CARDCOUNT, 7) || ' |' || BUFFER || '|';! FLUSH_CARDS
63 !               END;                                                     ! FLUSH_CARDS
64 !             END;                                                       ! FLUSH_CARDS
65 !           END;                                                         ! FLUSH_CARDS
66 !         END;                                                           ! FLUSH_CARDS
67 !       END;                                                             ! FLUSH_CARDS
68 !     END_PROC FLUSH_CARDS;                                              !
69 !                                                                        !
70 !   BUFFER = IFORMAT(10, 5);                                             !
71 !   BUFFER = BUFFER || PAD('     IS TEN', 10);                           !
72 !   CALL FLUSH_CARDS;                                                    !
73 !EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF !
```

```
39 CARDS WERE CHECKED.
NO ERRORS WERE DETECTED IN SCAN PHASE.
NO ERRORS WERE DETECTED IN PARSE PHASE.
MAXIMUM SYMBOL TABLE ENTRIES ARE 14  (LIMIT IS 500)
MAXIMUM MACRO TABLE ENTRIES ARE 4  (LIMIT IS 511)
MAXIMUM STRING TABLE ENTRIES ARE 71  (LIMIT IS 700)
MAXIMUM STRING TABLE LENGTH IS 483  (LIMIT IS 6500)
NUMBER OF STRING ENTRIES IS 221 NUMBER OF COLLISIONS IS 173
COMPILER IMPLEMENTATION RESTRICTS CONSTANT STRINGS TO 256 CHARACTERS.
MAXIMUM GRAPH STACK SIZE IS 98  (LIMIT IS 1600)
WORDS OF CODE GENERATED = 512B
TOTAL STORAGE NEEDED INCLUDING MONITOR IS 3556B WORDS
NUMBER OF BLOCKS IN OBJECT FILE IS 2
NUMBER OF BUFFER SWAP REQUESTS WAS 7
NUMBER OF CODE BUFFERS INPUT/OUTPUT = 2


   11    X80                        CHARACTER SCALAR                 AT 3151(0)
                                       27, 21
   12    CARDCOUNT                  FIXED     SCALAR                 AT 3162(0)
                                       35, 33, 32, 32
   13    BUFFER                     CHARACTER SCALAR                 AT 3163(0)
                                       38, 38, 38, 35, 33, 32, 31, 31
   14    CONTROL                    FIXED     VECTOR  (   64 )       AT 3174(0)
                                       34, 33, 31, 30
   19    PAD                        CHARACTER PROCEDURE              AT 3323(0)
                                       38
   20       PARAMETER               CHARACTER SCALAR                 AT 3300(0)
   19       PARAMETER               FIXED     SCALAR                 AT 3276(0)
   23    IFORMAT                    CHARACTER PROCEDURE              AT 3375(0)
                                       38, 35, 33
   23       PARAMETER               FIXED     SCALAR                 AT 3347(0)
   23       PARAMETER               FIXED     SCALAR                 AT 3350(0)
   30    FLUSH_CARDS                FIXED     PROCEDURE              AT 3442(0)
                                       39
```

```
MACRO DEFINITIONS:

NAME                   TEXT

STRINGLIMIT            160
                           24, 23, 20, 19
THENN                  THEN DO;
                           34, 33, 32, 31, 27, 20
ELSEE                  END;ELSE DO;
                           34
```

```
************  SUMMARY OF ERRORS REPAIRED.  ************

CARD NO.   LINE NO.    ERROR MESSAGES AND CORRECTIONS.
________   ________    _______________________________
```

TEST PROGRAM NO. 6

```
DECLARE CASEE0  LITERALLY 'DO;/*' ,CASEE LITERALLY 'END;DO;/*' ,
END_CASE LITERALLY 'END;END' , LOWER_LIMIT LITERALLY '0' ,
UPPER_LIMIT LITERALLY '5' , ITER_LIMIT LITERALLY '50' ;

DECLARE I FIXED ,ZEROS FIXED INITIAL (0),ONES FIXED INITIAL (0),
TWOS FIXED INITIAL (0), THREES FIXED INITIAL (0),
FOURS FIXED INITIAL (0), FIVES FIXED INITIAL (0),
STARTING POINT FIXED INITIAL(3571);
RANDOM_NO:    /* GENERATES RANDOM NUMBERS BETWEEN GIVEN LIMITS */
PROCEDURE;DECLARE K FIXED;K=15625*STARTING_POINT+22221;
STARTING_POINT=K-(K/2147483648)*2147483648;
RETURN LOWER_LIMIT+UPPER_LIMIT*STARTING_POINT/2147483648;
END_PROC RANDOM_NO;NEXT_NUMBER:PROCEDURE;DO CASE RANDOM_NO;
CASEE0:      /* ZERO */
ZEROS=ZEROS+1;

CASEE(1):    /* ONE */
ONES=ONES+1;

CASEE(2):    /* TWO */
TWOS=TWOS+1;

CASEE(3):    /* THREE */
THREES=THREES+1;

CASEE(4):    /* FOUR */
FOURS=FOURS+1;

CASEE(5):    /* FIVE */
FIVES=FIVES+1;

END_CASE;END_PROC NEXT_NUMBER;

DO I=0 TO ITER_LIMIT;CALL NEXT_NUMBER;END;

OUTPUT='ZEROS   = ' ||ZEROS;OUTPUT='ONES   = '||ONES;
OUTPUT='TWOS    = ' ||TWOS ;OUTPUT='THREES = '||THREES;
OUTPUT='FOURS   = ' ||FOURS;OUTPUT='FIVES  = '||FIVES;EOF EOF
```

CONCORDIA UNIVERSITY - X P L COMPILER - VERSION 5.0

```
  1 !                                                                          !
  2 !  DECLARE CASEE0 LITERALLY 'DO;/*',                                       !
  3 !          CASEE LITERALLY 'END;DO;/*',                                    !
  4 !          END_CASE LITERALLY 'END;END',                                   !
  5 !          LOWER_LIMIT LITERALLY '0',                                      !
  6 !          UPPER_LIMIT LITERALLY '5',                                      !
  7 !          ITER_LIMIT LITERALLY '50';                                      !
  8 !                                                                          !
  9 !  DECLARE I FIXED,                                                        !
 10 !          ZEROS FIXED INITIAL(0),                                         !
 11 !          ONES FIXED INITIAL(0),                                          !
 12 !          TWOS FIXED INITIAL(0),                                          !
 13 !          THREES FIXED INITIAL(0),                                        !
 14 !          FOURS FIXED INITIAL(0),                                         !
 15 !          FIVES FIXED INITIAL(0),                                         !
 16 !          STARTING_POINT FIXED INITIAL(3571);                             !
 17 !                                                                          !
 18 !                                                                          !
 19 !                                                                          !
 20 !                                                                          !
 21 !  RANDOM_NO:   /* GENERATES RANDOM NUMBERS BETWEEN GIVEN LIMITS */        !
 22 !     PROCEDURE;                                                           !
 23 !        DECLARE K FIXED;                                                  ! RANDOM_NO
 24 !        K = 15625*STARTING_POINT + 22221;                                 ! RANDOM_NO
 25 !        STARTING_POINT = K - (K/2147483648)*2147483648;                   ! RANDOM_NO
 26 !        RETURN LOWER_LIMIT + UPPER_LIMIT*STARTING_POINT/2147483648;       ! RANDOM_NO
 27 !     END_PROC RANDOM_NO;                                                  !
 28 !                                                                          !
 29 !                                                                          !
 30 !  NEXT_NUMBER:                                                            !
 31 !     PROCEDURE;                                                           !
 32 !        DO CASE RANDOM_NO;                                                ! NEXT_NUMBER
 33 !           CASEE0:      /* ZERO */                                        ! NEXT_NUMBER
 34 !              ZEROS = ZEROS + 1;                                          ! NEXT_NUMBER
 35 !                                                                          ! NEXT_NUMBER
 36 !           CASEE(1):    /* ONE */                                         ! NEXT_NUMBER
 37 !              ONES = ONES + 1;                                            ! NEXT_NUMBER
 38 !                                                                          ! NEXT_NUMBER
 39 !           CASEE(2):    /* TWO */                                         ! NEXT_NUMBER
 40 !              TWOS = TWOS + 1;                                            ! NEXT_NUMBER
 41 !                                                                          ! NEXT_NUMBER
 42 !           CASEE(3):    /* THREE */                                       ! NEXT_NUMBER
 43 !              THREES = THREES + 1;                                        ! NEXT_NUMBER
 44 !                                                                          ! NEXT_NUMBER
 45 !           CASEE(4):    /* FOUR */                                        ! NEXT_NUMBER
 46 !              FOURS = FOURS + 1;                                          ! NEXT_NUMBER
 47 !                                                                          ! NEXT_NUMBER
 48 !           CASEE(5):    /* FIVE */                                        ! NEXT_NUMBER
 49 !              FIVES = FIVES + 1;                                          ! NEXT_NUMBER
 50 !                                                                          ! NEXT_NUMBER
 51 !           END_CASE;                                                      ! NEXT_NUMBER
 52 !     END_PROC NEXT_NUMBER;                                                !
```

```

   DO I = 0 TO ITER_LIMIT;
      CALL NEXT_NUMBER;
   END;

   OUTPUT = 'ZEROS  = ' || ZEROS;
   OUTPUT = 'ONES   = ' || ONES;
   OUTPUT = 'TWOS   = ' || TWOS;
   OUTPUT = 'THREES = ' || THREES;
   OUTPUT = 'FOURS  = ' || FOURS;
   OUTPUT = 'FIVES  = ' || FIVES;
EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF
```

```
38 CARDS WERE CHECKED.
NO ERRORS WERE DETECTED IN SCAN PHASE.
NO ERRORS WERE DETECTED IN PARSE PHASE.
MAXIMUM SYMBOL TABLE ENTRIES ARE 12  (LIMIT IS 500)
MAXIMUM MACRO TABLE ENTRIES ARE 7  (LIMIT IS 511)
MAXIMUM STRING TABLE ENTRIES ARE 81  (LIMIT IS 700)
MAXIMUM STRING TABLE LENGTH IS 557  (LIMIT IS 6500)
NUMBER OF STRING ENTRIES IS 179 NUMBER OF COLLISIONS IS 154
COMPILER IMPLEMENTATION RESTRICTS CONSTANT STRINGS TO 256 CHARACTERS.
MAXIMUM GRAPH STACK SIZE IS 85  (LIMIT IS 1600)
WORDS OF CODE GENERATED = 310B
TOTAL STORAGE NEEDED INCLUDING MONITOR IS 3354B WORDS
NUMBER OF BLOCKS IN OBJECT FILE IS 2
NUMBER OF BUFFER SWAP REQUESTS WAS 3
NUMBER OF CODE BUFFERS INPUT/OUTPUT = 2


     5    I                                FIXED     SCALAR           AT 3151(0)
                                             34
     5    ZEROS                            FIXED     SCALAR           AT 3152(0)
                                             36, 15, 15
     5    ONES                             FIXED     SCALAR           AT 3153(0)
                                             36, 18, 18
     6    TWOS                             FIXED     SCALAR           AT 3154(0)
                                             37, 21, 21
     6    THREES                           FIXED     SCALAR           AT 3155(0)
                                             37, 24, 24
     7    FOURS                            FIXED     SCALAR           AT 3156(0)
                                             38, 27, 27
     7    FIVES                            FIXED     SCALAR           AT 3157(0)
                                             38, 30, 30
     8    STARTING_POINT                   FIXED     SCALAR           AT 3160(0)
                                             12, 11, 10
    10    RANDOM_NO                        FIXED     PROCEDURE        AT 3167(0)
                                             13
    13    NEXT_NUMBER                      FIXED     PROCEDURE        AT 3211(0)
                                             34
```

```
MACRO DEFINITIONS:

NAME                    TEXT

CASEEO                  DO;/*
                          14
CASEE                   END;DO;/*
                          29, 26, 23, 20, 17
END_CASE                END;END
                          32
LOWER_LIMIT             0
                          12
UPPER_LIMIT             5
                          12
ITER_LIMIT              50
                          34
```

```
************  SUMMARY OF ERRORS REPAIRED.  ************

CARD NO.   LINE NO.    ERROR MESSAGES AND CORRECTIONS.
________   ________    _______________________________
```

TEST PROGRAM NO. 7

```
/*    $L    */

/*****************************/
/*   FORMAT PROCEDURES      */
/*****************************/

DECLARE   STRINGLIMIT    LITERALLY  '160',
 THENN  LITERALLY 'THEN DO;' ,
 ELSEE LITERALLY 'END ELSE DO ' ,
         X80          CHARACTER(80) INITIAL('                        '),
          CARDCOUNT     FIXED   INITIAL(0),
          BUFFER         CHARACTER(80),
          CONTROL(64)    FIXED;


PAD:    /*  TO PAD THE STRING */ PROCEDURE(STRING,WIDTH CHARACTER(
STRINGLIMIT ;DECLARE (NUMBER,WIDTH,L FIXED,STRING CHARACTER
(STRINGLIMIT  L=LENGTH(STRING  IF L<WIDTH THENN STRING
=SUBSTR(X80,0,WIDTH-L)||STRING;END;RETURN STRING; /* IT IS DONE */
END_PROC PAD;IFORMAT;  /* TO PAD NUMBERS */ PROCEDURE(NUMBER,WIDTH)
CHARACTER(STRINGLIMIT);DECLARE  (NUMBER,WIDTH,L)FIXED,
STRING CHARACTER(STRINGLIMIT);

STRING = NUMBER;  /* CHANGE NUMBER TO STRING */
L=LENGTH(STRING);IF L<WIDTH THENN STRING=SUBSTR(X80,0,WIDTH-L)
||STRING;    RETURN STRING; /* IT IS DONE */  END_PROC IFORMAT;
FLUSH_CARDS:  PROCEDURE;  /*  FLUSHES THE CARDS */
OUTPUT='***** FLUSH PROGRAM *****';IF CONTROL(BYTE('L')) OR
CONTROL(BYTE('M'))THENN DO WHILE LENGTH(BUFFER)NE 0;BUFFER=INPUT;
CARDCOUNT=CARDCOUNT+1;IF LENGTH(BUFFER)>0THENN IF
CONTROL(BYTE('M'))THENN OUTPUT=IFORMAT(CARDCOUNT,7)||' |'||BUFFER;
ELSEE IF CONTROL(BYTE('L'))THENN
OUTPUT=IFORMAT(CARDCOUNT,7)||' |'||BUFFER||'|';
END_PROC FLUSH_CARDS;

BUFFER=IFORMAT(10,5);BUFFER=BUFFER||PAD('     IS TEN',10);
CALL FLUSH_CARDS;EOF EOF EOF EOF
```

CONCORDIA UNIVERSITY - X P L COMPILER - VERSION 5.0

```
 !                                                        !
 !                                                        !
 !                                                        !
 !   /*    $L     */                                      !
 !                                                        !
 ! /*****************************/                        !
 ! /*    FORMAT PROCEDURES      */                        !
 ! /*****************************/                        !
 !                                                        !
 !  DECLARE STRINGLIMIT LITERALLY '160',                  !
 !          THENN LITERALLY 'THEN DO;',                   !
 !          ELSEE LITERALLY 'END ELSE DO;',               !
 !          X80 CHARACTER(80) INITIAL('              '),  !
 !          CARDCOUNT FIXED INITIAL(0),                   !
 !          BUFFER CHARACTER(80),                         !
 !          CONTROL(64) FIXED;                            !
 !                                                        !
 !                                                        !
 !                                                        !
 !  PAD:    /*  TO PAD THE STRING */                      !
 !     PROCEDURE(STRING, WIDTH) CHARACTER( STRINGLIMIT);  !
 !        DECLARE (NUMBER,                                ! PAD
 !                 WIDTH,                                 ! PAD
 !                 L) FIXED,                              ! PAD
 !                 STRING CHARACTER( STRINGLIMIT);        ! PAD
 !        L = LENGTH(STRING);                             ! PAD
 !        IF L < WIDTH THENN                              ! PAD
 !           STRING = SUBSTR(X80, 0, WIDTH - L) || STRING;! PAD
 !        END;                                            ! PAD
 !        RETURN STRING; /* IT IS DONE */                 ! PAD
 !     END_PROC PAD;                                      !
 !                                                        !
 !                                                        !
 !  IFORMAT; /* TO PAD NUMBERS */                         !
 !     PROCEDURE(NUMBER, WIDTH) CHARACTER( STRINGLIMIT);  !
 !        DECLARE (NUMBER,                                ! IFORMAT
 !                 WIDTH,                                 ! IFORMAT
 !                 L) FIXED,                              ! IFORMAT
 !                 STRING CHARACTER( STRINGLIMIT);        ! IFORMAT
 !                                                        ! IFORMAT
 !        STRING = NUMBER;  /* CHANGE NUMBER TO STRING */ ! IFORMAT
 !        L = LENGTH(STRING);                             ! IFORMAT
 !        IF L < WIDTH THENN                              ! IFORMAT
 !           STRING = SUBSTR(X80, 0, WIDTH - L) || STRING;! IFORMAT
 !           RETURN STRING; /* IT IS DONE */              ! IFORMAT
 !        END;                                            ! IFORMAT
 !     END_PROC IFORMAT;                                  !
```

```
48 !                                                                              !
49 !                                                                              !
50 !   FLUSH_CARDS:                                                               !
51 !      PROCEDURE;  /*  FLUSHES THE CARDS */                                     !
52 !         OUTPUT = '***** FLUSH PROGRAM *****';                                 ! FLUSH_CARDS
53 !         IF CONTROL(BYTE('L')) OR CONTROL(BYTE('M')) THENN                     ! FLUSH_CARDS
54 !            DO WHILE LENGTH(BUFFER) NE 0;                                      ! FLUSH_CARDS
55 !               BUFFER = INPUT;                                                 ! FLUSH_CARDS
56 !               CARDCOUNT = CARDCOUNT + 1;                                      ! FLUSH_CARDS
57 !               IF LENGTH(BUFFER) > 0 THENN                                     ! FLUSH_CARDS
58 !                  IF CONTROL(BYTE('M')) THENN                                  ! FLUSH_CARDS
59 !                     OUTPUT = IFORMAT(CARDCOUNT, 7) || ' |' || BUFFER;         ! FLUSH_CARDS
60 !                  ELSEE;                                                       ! FLUSH_CARDS
61 !                     IF CONTROL(BYTE('L')) THENN                               ! FLUSH_CARDS
62 !                        OUTPUT = IFORMAT(CARDCOUNT, 7) || ' |' || BUFFER || '|';! FLUSH_CARDS
63 !                     END;                                                      ! FLUSH_CARDS
64 !                  END;                                                         ! FLUSH_CARDS
65 !               END;                                                            ! FLUSH_CARDS
66 !            END;                                                               ! FLUSH_CARDS
67 !         END;                                                                  ! FLUSH_CARDS
68 !      END_PROC FLUSH_CARDS;                                                    !
69 !                                                                              !
70 !   BUFFER = IFORMAT(10, 5);                                                   !
71 !   BUFFER = BUFFER || PAD('     IS TEN', 10);                                  !
72 !   CALL FLUSH_CARDS;                                                          !
73 !EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF !
```

```
39 CARDS WERE CHECKED.
22 ERRORS WERE DETECTED AND REPAIRED IN SCAN PHASE.
NO ERRORS WERE DETECTED IN PARSE PHASE.
MAXIMUM SYMBOL TABLE ENTRIES ARE 14  (LIMIT IS 500)
MAXIMUM MACRO TABLE ENTRIES ARE 4  (LIMIT IS 511)
MAXIMUM STRING TABLE ENTRIES ARE 71  (LIMIT IS 700)
MAXIMUM STRING TABLE LENGTH IS 483  (LIMIT IS 6500)
NUMBER OF STRING ENTRIES IS 215 NUMBER OF COLLISIONS IS 161
COMPILER IMPLEMENTATION RESTRICTS CONSTANT STRINGS TO 256 CHARACTERS.
MAXIMUM GRAPH STACK SIZE IS 98  (LIMIT IS 1600)
WORDS OF CODE GENERATED = 512B
TOTAL STORAGE NEEDED INCLUDING MONITOR IS 3556B WORDS
NUMBER OF BLOCKS IN OBJECT FILE IS 2
NUMBER OF BUFFER SWAP REQUESTS WAS 7
NUMBER OF CODE BUFFERS INPUT/OUTPUT = 2


     11   X80                      CHARACTER SCALAR                   AT 3151(0)
                                     27, 21
     12   CARDCOUNT                FIXED     SCALAR                   AT 3162(0)
                                     35, 33, 32, 32
     13   BUFFER                   CHARACTER SCALAR                   AT 3163(0)
                                     38, 38, 38, 35, 33, 32, 31, 31
     14   CONTROL                  FIXED     VECTOR  (   64 )         AT 3174(0)
                                     34, 33, 31, 30
     19   PAD                      CHARACTER PROCEDURE                AT 3323(0)
                                     38
     20      PARAMETER             CHARACTER SCALAR                   AT 3300(0)
     19      PARAMETER             FIXED     SCALAR                   AT 3276(0)
     23   IFORMAT                  CHARACTER PROCEDURE                AT 3375(0)
                                     38, 35, 33
     23      PARAMETER             FIXED     SCALAR                   AT 3347(0)
     23      PARAMETER             FIXED     SCALAR                   AT 3350(0)
     30   FLUSH_CARDS              FIXED     PROCEDURE                AT 3442(0)
                                     39
```

```
MACRO DEFINITIONS:

NAME                    TEXT

STRINGLIMIT             160
                           24, 23, 20, 19
THENN                   THEN DO;
                           34, 33, 32, 31, 27, 20
ELSEE                   END ELSE DO
                           34
```

```
MACRO DEFINITIONS$

NAME              TEXT

STRINGLIMIT       160
                      24, 23, 20, 19
THENN             THEN DO;
                      34, 33, 32, 31, 27, 20
ELSEE             END ELSE DO
                      34
```

```
************  SUMMARY OF ERRORS REPAIRED.  ************

CARD NO.  LINE NO.   ERROR MESSAGES AND CORRECTIONS.
________  ________   _______________________________

   18        21      *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                     *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   )   CHARACTER .

   19        21      *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                     *** CORRECTED SYMBOL SEQUENCE IS $   <NUMBER>   )   ; .

   19        24      *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                     *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   )   FIXED .

   20        25      *** ILLEGAL SYMBOL PAIR $   <NUMBER>   <IDENTIFIER> .

   20        25      *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                     *** CORRECTED SYMBOL SEQUENCE IS $   <NUMBER>   )   <IDENTIFIER> .

   20        25      *** ILLEGAL SYMBOL PAIR $   )   <IDENTIFIER> .
                     *** CORRECTED SYMBOL SEQUENCE IS $   )   ;   <IDENTIFIER> .

   20        26      *** ILLEGAL SYMBOL PAIR $   <IDENTIFIER>   IF .

   20        26      *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                     *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   )   IF .

   20        26      *** ILLEGAL SYMBOL PAIR $   )   IF .
                     *** CORRECTED SYMBOL SEQUENCE IS $   )   ;   IF .

   22        32      *** ILLEGAL SYMBOL PAIR $   ;   PROCEDURE .
                     ***  ;   IS REPLACED BY   $ .

   28        45      *** ERROR$  UNMATCHED 'DO' .
                     *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .

   28        46      *** ILLEGAL SYMBOL PAIR $   END   END_PROC .
                     *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   END_PROC .

   34        60      *** ILLEGAL SYMBOL PAIR $   END   ELSE .
                     *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   ELSE .

   34        60      *** ILLEGAL SYMBOL PAIR $   DO   IF .
                     *** CORRECTED SYMBOL SEQUENCE IS $   DO   ;   IF .

   36        62      *** ERROR$  UNMATCHED 'DO' .
                     *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .

   36        63      *** ILLEGAL SYMBOL PAIR $   END   END_PROC .
                     *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   END_PROC .
```

```
************ SUMMARY OF ERRORS REPAIRED. ************

CARD NO.   LINE NO.    ERROR MESSAGES AND CORRECTIONS.
________   ________    _______________________________

   18         21       *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                       *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   )   CHARACTER .

   19         21       *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                       *** CORRECTED SYMBOL SEQUENCE IS $   <NUMBER>   )   ; .

   19         24       *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                       *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   )   FIXED .

   20         25       *** ILLEGAL SYMBOL PAIR $   <NUMBER>   <IDENTIFIER> .

   20         25       *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                       *** CORRECTED SYMBOL SEQUENCE IS $   <NUMBER>   )   <IDENTIFIER> .

   20         25       *** ILLEGAL SYMBOL PAIR $   )   <IDENTIFIER> .
                       *** CORRECTED SYMBOL SEQUENCE IS $   )   ;   <IDENTIFIER> .

   20         26       *** ILLEGAL SYMBOL PAIR $   <IDENTIFIER>   IF .

   20         26       *** ERROR$  UNMATCHED LEFT PARENTHESIS.
                       *** CORRECTED SYMBOL SEQUENCE IS $   <IDENTIFIER>   )   IF .

   20         26       *** ILLEGAL SYMBOL PAIR $   )   IF .
                       *** CORRECTED SYMBOL SEQUENCE IS $   )   ;   IF .

   22         32       *** ILLEGAL SYMBOL PAIR $   ;   PROCEDURE .
                       ***  ;   IS REPLACED BY   $ .

   28         45       *** ERROR$  UNMATCHED 'DO' .
                       *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .

   28         46       *** ILLEGAL SYMBOL PAIR $   END   END_PROC .
                       *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   END_PROC .

   34         60       *** ILLEGAL SYMBOL PAIR $   END   ELSE .
                       *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   ELSE .

   34         60       *** ILLEGAL SYMBOL PAIR $   DO   IF .
                       *** CORRECTED SYMBOL SEQUENCE IS $   DO   ;   IF .

   36         62       *** ERROR$  UNMATCHED 'DO' .
                       *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .

   36         63       *** ILLEGAL SYMBOL PAIR $   END   END_PROC .
                       *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   END_PROC .
```

```
36      63      *** ERROR$  UNMATCHED 'DO' .
                *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .
36      64      *** ILLEGAL SYMBOL PAIR $  END   END_PROC .
                *** CORRECTED SYMBOL SEQUENCE IS $  END   ;   END_PROC .
36      64      *** ERROR$  UNMATCHED 'DO' .
                *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .
36      65      *** ILLEGAL SYMBOL PAIR $  END   END_PROC .
                *** CORRECTED SYMBOL SEQUENCE IS $  END   ;   END_PROC .
36      65      *** ERROR$  UNMATCHED 'DO' .
                *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .
36      66      *** ILLEGAL SYMBOL PAIR $  END   END_PROC .
                *** CORRECTED SYMBOL SEQUENCE IS $  END   ;   END_PROC .
36      66      *** ERROR$  UNMATCHED 'DO' .
                *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .
36      67      *** ILLEGAL SYMBOL PAIR $  END   END_PROC .
                *** CORRECTED SYMBOL SEQUENCE IS $  END   ;   END_PROC
```

```
36      63      *** ERROR$  UNMATCHED 'DO' .
                *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .

36      64      *** ILLEGAL SYMBOL PAIR $   END   END_PROC .
                *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   END_PROC .

36      64      *** ERROR$  UNMATCHED 'DO' .
                *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .

36      65      *** ILLEGAL SYMBOL PAIR $   END   END_PROC .
                *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   END_PROC .

36      65      *** ERROR$  UNMATCHED 'DO' .
                *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .

36      66      *** ILLEGAL SYMBOL PAIR $   END   END_PROC .
                *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   END_PROC .

36      66      *** ERROR$  UNMATCHED 'DO' .
                *** CORRECTED SYMBOL SEQUENCE IS $   ;   END   END_PROC .

36      67      *** ILLEGAL SYMBOL PAIR $   END   END_PROC .
                *** CORRECTED SYMBOL SEQUENCE IS $   END   ;   END_PROC .
```

BIBLIOGRAPHY

AH1 Aho, A.V.; Ullman, J.D.: The theory of parsing, translation and compiling. Volume I. Parsing, Prentice-Hall series in automatic computation. 1972.

AH2 Aho, A.V.; Johnson, S.C.: LR Parsing. Computing Surveys 2, 6 (June 1974), pp 99-124.

BA1 Barnard, D.T.: Automatic generation of syntax-repairing and paragraphing parsers. Technical Report CSRG-52, Computer Systems Research Group, University of Toronto, Ontario, April 1975.

BO1 Bochmann, G.V.: Attribute grammars and compilation:- Program evaluation in several phases: University of Montreal, Quebec, August 1974.

BR1 Brooker, R.A.; MacCallum, I.R.; Morris, D.; Rohl, J.S.: The compiler-compiler. Annual Review in Automatic Programming, Vol. 3, 1963, pp 229-276.

CL1 Clark, B.J.; Ham, F.J.B.: The project SUE system language reference manual. Technical Report CSRG-42, Computer Systems Research Group, University of Toronto, Ontario, September 1974.

CO1 Cordy, J.R.: A diagramatic approach to programming language semantics. Technical Report CSRG-67, Computer Systems Research Group, University of Toronto, Ontario, March 1976.

CO2 Conway, R.W.; Wilcox, T.R.: Design and implementation of a diagnostic compiler for PL/I. Comm. ACM 16, 3 (March 1973), pp 169-179.

DR1 DeRemer, F.L.: Practical translators for LR(K) languages: Ph.D. Dissertation, MIT, October 1969.

FE1 Feldman, J.A.: A formal semantics for computer languages and its application in a compiler-compiler. Comm. ACM 9, 1 (January 1966), pp 3-9.

FE2 Feldman, J.A.; Gries, D.: Translator Writing Systems. Comm. ACM 11, 2 (February 1968), pp 77-108.

GO1 Gorrie, J.D.: A processor generator system. Technical Report CSRG-3, Computer Systems Research Group, University of Toronto, Ontario, Feburary 1971.

GR1 Griffiths, M.: Introduction to compiler compilers, in Compiler Construction: An Advanced Course, Ed. Bauer, F.L., Eickel, J., Springer-Verlag, 1974.

GR2 Graham, S.L.; Rhodes, S.P.: Practical syntactic error recovery in compilers. Proceedings of ACM symposium on Priciples of Programming Languages. Boston 1973.

HO1 Holt, R.C.; Wortman, D.B.; Barnard, D.T.; Cordy, J.R.: SP/k: A system for teaching computer programming. Comm. ACM 20, 5 (May 1977), pp 301-309.

IBM IBM: OS PL/I checkout compiler programmer's guide, SC33-0007.

IR1 Irons, E.T.: An error-correcting parse algorithm. Comm. ACM 6, 11 (November 1963), pp 669-673.

JAI James, L.R.: A syntax directed error recovery method. Technical Report CSRG-13. Computer Systems Research Group, University of Toronto, Ontario. May 1972.

KN1 Knuth, D.E.: Backus Normal Form vs Backus Naur Form. Comm. ACM 7, 12 (December 1964), pp 735-736.

KN2 Knuth, D.E.: Computer programming as an art. Comm. ACM 17, 12 (December 1974), pp 667-673.

LE1 Lecarme, O.; Bochmann, G.V.: A compiler writing system - user's manual. Universite de Montreal, December 1974.

LE2 Leinius, R.P.: Error detection and recovery for syntax directed compiler systems. Ph.D. thesis, University of Wisconsin, 1970.

LI1 Litecky, R.; Davis, B.: A study of errors, error-proneness, and error diagnosis in Cobol. Comm. ACM 19, 1 (January 1976), pp 33-37.

ME1 Meyer, A.R.; Ritchie, D.M.: The compexity of LOOP programs. Proceeding 22nd ACM National Conference, pp 465-469.

MK1 McKeeman, W.M.; Horning, J.J.; Wortman, D.B.: A compiler generator. Printice-Hall, Inc., Englewood Cliffs, N. J., 1970.

MO1 Morgan, H.L.: Spelling correction in system programs. Comm. ACM 13, 2 (February 1970), pp 90-9.

RE1 Reynolds, J.C.: GADANKEN - a simple typeless language based on the principle of completeness and the reference concept. Comm. ACM 13, 5 (May 1970), pp 308-319.

RU1 Rudmik, A.: On the generation of optimizing compilers. Ph.D. thesis, University of Toronto, 1975.

SC1 Scott, D.; Strachey, C.: Towards a mathematical semantics for computer languages. Proc. Symp. On Computers and Automata, Polytechnic Institute of Brooklyn. 1971.

TE1 Tennent, R.D.: The denotational semantics of programming languages. Comm. ACM 19, 8 (August 1976), pp 437-453.

WI1 Wirth, N.: PL360, a programming language for 360 computers. JACM 15, 1 (March 1973), pp 37-74.