National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

NL-339 (r.86/06)

Canada

A Translator for the Multiprocessing Language Pascal-C

Stephen Cabilio

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
Montréal, Québec, Canada

March 1986

© Stephen Cabilio, 1986

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-30637-8

ABSTRACT

A Translator for the Multiprocessing Language Pascal-C

Stephen Cabilio

This thesis describes a translator for the Pascal-C programming language. Pascal-C is an extension of Pascal for solving combinatorial and similar computationally-bound problems on a loosely-coupled multiprocessor. This language is one component of a prototype Pascal-C system developed at Concordia. The underlying hardware is inexpensive and easily assembled, consisting of micro- and mini-processors connected by serial lines. The most prominent characteristic of the system is its hierarchical, master/slave architecture which suits the decomposable nature of the type of problem it is intended to solve. The powerful high-level constructs in Pascal-C allow the applications programmer to exploit the master/slave concurrency in the system with relative ease. The description of the translator is organized according to the main phases of translation, with special attention to difficult areas such as dealing systematically with syntactic errors. Also described are some software tools used to develop and maintain the translator, and finally the development of an adequate scheduling mechanism to support concurrent processes in the Pascal-C run-time system.

ACKNOWLEDGEMENTS

My warmest thanks to my supervisor Dr. J. Opatrny, who was always generous with his time and knowledge. His advice and encouragement were vital to the preparation of this thesis. I am also grateful to the other members of the Pascal-C group for the many thought-provoking discussions that went into the design of this system, and in particular to Dr. C. Lam and L. Thiel for their personal involvement and ingenious suggestions. My thanks also to Dr. D. Ford and P. Dubois, who on several occasions kindly assisted me when their technical expertise was most welcome.

TABLE OF CONTENTS

1. Introduction

There are many problems in combinatorial mathematics that are simple to state and have obvious solutions, but nevertheless require a great deal of computation. A classic example is the knapsack problem. In one version of this problem, we are given a set S of integers and a separate integer t, and we are asked to determine if any combination of the elements in S adds up to t. This can of course be solved by attempting the summation for every subset of S until either one is found that satisfies the criterion or else all subsets have been tried unsuccessfully. It is also clear that this method is highly inefficient, since the number of subsets of S is 2 to the power of |S| (the size of S). Yet there is no alternative known that is substantially better than this crude method; that is, for all known algorithms the worst-case computing time is an exponential function of |S|.

The knapsack problem is typical of a large class known as NP-complete problems [1], many of which are of keen interest, both pragmatic and theoretical, in a wide variety of disciplines. For all these problems, the only exact algorithms known must consider many possibilities, whose number increases exponentially with the size of the data given to the problem. It is widely suspected, though not yet proved, that there cannot in fact exist any substantially better method. If so, this places a severe limit on the size of any problem instance for which a solution can be expected in a reasonable length of time, even with the fastest computing machines now available. As for the future, although the technological improvements over the years have been remarkable, the increase in computing speed has been essentially linear, so the apparent intractability of these problems is unlikely to be relieved by brute force. One cannot hope for more than a modest increase in the size of the problem instances that are practical to solve. This is nevertheless a worthwhile goal, since some problem instances that are of interest to combinatorial mathematicians (and others) seem to be just out of reach at present.

The fastest computers are very expensive, but they provide facilities that are not needed for the kind of problem under consideration, such as sophisticated multiple-user operating systems, and hardware for extremely fast floating-point arithmetic. The mini- and microcomputers, on the other hand, are dropping dramatically in price though they remain rather slow. The prospect of combining the computing power of several small processors is becoming more and more appealing, since if this could be done effectively for the problems at hand, the power of a large computer would be obtained at a fraction of the cost.

Indeed, problems like knapsack appear to be particularly well-suited for multi-microprocessors. Any large instance of the problem can be expressed in terms of slightly smaller instances, as is evident from the recursive algorithms often employed for such problems. If s is a member of S in the knapsack example, the problem may be reformulated as two subproblems, both with S - {s} as the set, and one with t as the target and the other with t - s. The first subproblem considers the possibilities with s excluded from the solution subset; the second with s included. If the answer to either is positive, then so is the answer to the larger problem. This is equivalent to searching a tree in which the nodes represent subproblems that are decomposed into smaller and more numerous subproblems at each successive level. The subtrees of sibling nodes are disjoint, which indicates that the corresponding subproblems are independent in terms of data and order of evaluation. In other words, such subproblems can be solved simultaneously with no communication among them. The parent's communication with its children, moreover, is limited to sending them data at the beginning and receiving their results at the end. Even this movement of data is expected to be small in comparison with the amount of computation required, except perhaps at the lowest levels of the tree.

The architecture suggested by these observations is a hierarchical one, wherein a master processor generates subproblems of suitable size and submits them for solution on slave processors, which work in parallel and return their results to the master at the end. Compared with a more general configuration, a hierarchical arrangement has the advantage of requiring few logical interconnections (one less than the number of nodes). Moreover, like hierarchical structures in systems of any sort, such an organization is easier to understand and control, which is an important consideration in the complicated and error-prone business of parallel computation.

A prototype of such a single-user multiprocessor has been developed at Concordia [2]. It takes advantage of the low communication requirements between the nodes of the system by assembling inexpensive, off-the-shelf components into a loosely-coupled system. No exotic or specially designed hardware is required. At present, the prototype features a PDP-11/34 and two LSI-11/23's connected via serial lines. Each computer has access to peripherals such as monitors and disk drives, though this may not be true of processors added in the future. Note that the physical layout of the interconnections is not important, as long as the logical connections can be configured into a virtual hierarchy [3]. The current system supports a two-level hierarchy, also known as a star configuration, and will probably not support a multi-level hierarchy (with intermediate-level processors acting as both slaves to the higher level and masters to the lower level) for some time to come. The additional effort required to implement such a generalization would only be justified if there were too many processors for a single master to service them efficiently and keep them busy with subproblems.

Thus the goal of economy is easily satisfied, at least as far as the hardware is concerned. The more difficult goals, which depend largely on the software, are to ensure that the system's computing power is utilized effectively and that the system is reasonably simple to use for the intended applications. Accordingly, an important part of the project was the development of the Pascal-C language, an extension of standard Pascal [4] which enables the user to specify how the problem is to be decomposed into independent subproblems to be solved on the slave processors. The main subject of this thesis is the implementation of a translator of Pascal-C for this system. A closely related component, the language's run-time system (RTS for short), was implemented by Yin-Lam Wong and described in her thesis [5].

Chapter 2 reviews the Pascal-C language and chapter 3, the core of this thesis, describes its translator. Chapter 4 describes some software tools that helped in the implementation of the translator and can be used to maintain it. Finally, chapter 5 describes work done in connection with certain run-time features, principally with regard to process scheduling.

2. The Pascal-C Language

No automatic method has yet been developed for detecting the inherent parallelism in an algorithm and exploiting this knowledge effectively and reliably on a multiprocessor. A programming language must therefore provide special features that allow a programmer to communicate this knowledge of parallelism more or less explicitly. The existing concurrent languages (e.g., Concurrent Pascal [6]) tend to be oriented towards systems programming on a single processor rather than applications programming on a multiprocessor. It was therefore decided to devise a language that is appropriate for solving combinatorial problems such the knapsack problem.

As its name implies, Pascal-C is a variation of standard Pascal [4]. Rather than create an entirely new language, which would require a great deal of work for its design and implementation as well as a substantial learning effort on the part of prospective users, we extended a very familiar language with a few powerful constructs. If used properly, the new features enable the programmer to exploit the multiprocessor's computing potential without being burdened with irrelevant details concerning the system's operation. The succinctness of the language for the intended applications is forcefully illustrated when one compares a typical Pascal-C program with its labouriously coded equivalent in Concurrent Pascal [7]. Not surprisingly, the very power of these constructs results in elaborate semantic specifications that are in contrast with their concise appearance. This chapter describes Pascal-C's special features: the DOWN procedure, the COPY section, the CRITICAL procedure, the WAIT procedure, and the TERMINATE procedure. Appendix 1.1 shows a complete Pascal program to solve the knapsack problem, with a Pascal-C equivalent in appendix 1.2 for comparison. Similarily, appendices 2.1 and 2.2 are quicksort implementations written in Pascal and Pascal-C respectively. Taken together, the appendices illustrate all the special features of Pascal-C. The description in this chapter is informal, concentrating on some implications of Pascal-C's features for their use and implementation. A thorough discussion of the language's history and rationale can be found in the Pascal-C report [8].

2.1. The DOWN Procedure

As illustrated in appendix 1.2, a DOWN procedure looks like a regular procedure except for the keyword DOWN and the optional COPY section, which is discussed in section 2.2. Note the heading in the example:

```
DOWN PROCEDURE SLAVETRY (SUM : INTEGER; CHOICE : SUBSET;
            CRITICAL PROCEDURE SOLUTION (CHOICE : SUBSET));
```

The master-slave relationship between processors, which was described in the introductory chapter, is reflected in the static structure of a Pascal-C program. The parts of the program that are executed in the slaves are the DOWN procedures; all other parts are executed in the master. The DOWN procedure may be considered the most important of Pascal-C's special features, since its semantics cover the principal rules for parallel execution and data communication among processors. This static partitioning of the program into master and slave parts has its merits, since much of the analysis can thereby be done during the translation, but it also has disadvantages from the point of view of programming convenience. It means that a recursive pattern must be broken at some point to call the DOWN procedures. Typically, there is one type of recursive procedure declared for the master which calls the DOWN procedure at its leaves, and another recursive procedure declared for the slave which resembles its equivalent in a sequential program. The partitioning also implies that the program's logic must make good decisions concerning the appropriate points to call the DOWN procedures. Otherwise, the subproblems sent to the slaves may be too small or too large. If too small, the overhead will be excessive, especially with regard to communication. If too large, there is a chance that some slaves may be idle while others have long tasks, especially near the end of the computation.

DOWN procedures can be nested at an arbitrary level within regular procedures and functions, they may have any number of regular procedures and functions nested within them (which, like their host procedures, are executed in the slaves), and they are invoked exactly like regular procedures.

DOWN procedures are always called from the master part of the program, which implies that they may not call themselves or other DOWN procedures, either directly or indirectly. (In a multi-level hierarchy, however, a DOWN procedure could call another DOWN procedure nested within it, which would correspond to an intermediate-level processor activating a lower-level slave. In the current two-level implementation, such nesting is forbidden, and the ensuing discussion will take only a two-level system into account. For all features of the language, nevertheless, the extrapolation to a multi-level system is fairly straightforward.) The invocation of a DOWN procedure results in the dynamic creation of a new process, an activation of the DOWN procedure that will execute asynchronously in some available slave processor. In the meantime, the master is free to proceed beyond the invocation statement and continue generating processes, most typically (though not necessarily) activations of the same DOWN procedure with different subproblem data. The master is automatically blocked as long as there is no slave available for the current invocation, so the programmer need not worry about the number of processes being generated or the number of processors in the system. (The blocking could be forestalled by implementing a queue for requested activations, but the queue would be liable to fill up rapidly since subproblems tend to be generated more quickly than they can be solved. So blocking would soon occur in any case, and no significant effect on the total waiting time could be expected.)

The master can only send data to the slave at the time the DOWN procedure is invoked: namely, the actual parameters and the current values of whatever relatively global variables are listed in the COPY section. The language does not provide the master with any means for subsequently altering or adding to the original subproblem data while the slave is still executing. (Future extensions to the language may allow the master to update the slave's data periodically, as would be required for certain important applications such as branch-and-bound algorithms.) The evaluation of the parameters is treated as a critical region and is thereby protected from possibly disruptive actions from the slaves (see the CRITICAL procedure feature in section 2.3). If the termination condition on the DOWN procedure is set (see the TERMINATE feature in section 2.5), no activation will take place but the DOWN procedure's parameters will nevertheless be evaluated, so that any side effects caused by the evaluation will occur.

There are two quite different ways for the slave to return results. One way, through CRITICAL procedure calls, is described in section 2.3. The other is through the DOWN procedure's VAR parameters. In appendix 2.2, for example, we have the declaration

```
DOWN PROCEDURE SLAVESORT (VAR E : VECTOR; LO, HI : INTEGER);
```

which causes the contents of A to be returned to the master at the end of each activation of PARQUICK in the slaves. The VAR parameters in a DOWN procedure, like the value parameters, exist as local copies within the slave. The updating of the actual parameters in the master takes place only some time after the end of the DOWN procedure (see the WAIT feature, section 2.4), so the semantics are in this sense more like those of value-result parameters than those of regular VAR parameters in Pascal. Note that any changes to the variables listed in the COPY section affect only the slave's local copies, so results cannot be returned to the master by modifying relatively global data.

At some point in the program, the master must synchronize with the slaves by calling the WAIT procedure, which blocks the master until all processes connected with the given DOWN procedure have come to completion. Only then are the slave's VAR parameters, which until now have been buffered, used to update the actual parameters in the master. A novel aspect of this updating is that only those components of the VAR parameters that have actually been altered will be written back to the original variables. Thus, for example, a single array could be sent as a VAR parameter to several activations of the DOWN procedure, and the slaves could update different portions of the array.

This selective updating modifies the value-result semantics mentioned in the previous paragraph to simulate the properties of regular VAR parameters. Moreover, there is a run-time check to ensure that no component of such an actual VAR parameter has been altered by different activations of the DOWN procedure. The sets of components updated by the various activations must be disjoint, or else an error is reported. (In this context, "component" means all simple types and set types, since these are effectively irreducible. A component is judged to have been altered when its value at the end of the DOWN procedure is different from its value at the beginning of that procedure, regardless of any changes that may meanwhile have taken place in the original copy in the master.) It was reasoned that conflicting updates by the DOWN procedures could not possibly be part of the programmer's intention, since these are bound to overwrite each other in an unpredictable sequence. These rules apply also to simple VAR parameters, which are treated as single-component types.

There are a number of special restrictions regarding DOWN procedures, besides the ones already mentioned. Any relatively global object that is referenced in the body of the DOWN procedure must be identified in its COPY section (described in section 2.2). The standard predefined identifiers are available to the DOWN procedure without appearing in the COPY section, except for the special Pascal-C procedures WAIT and TERMINATE which cannot be invoked by a DOWN procedure. The data types that appear in the DOWN procedure's parameter list must not have any pointer or file components, since the different processors do not have access to common memory or peripherals. In addition, VAR parameters must not be part of dynamic variables, since there is no reasonable way to check whether or not they have been mistakenly disposed by the time the updating takes place. Record variants are also excluded within VAR parameters because component-by-component updating becomes problematic when different variants are involved. CRITICAL procedures are allowed as formal parameters to DOWN procedures, but regular procedures and functions are not. Finally, DOWN procedures are not allowed as parameters to any kind of procedure or function.

2.2. The COPY Section

The COPY section is an optional feature following the DOWN procedure heading. It consists of the keyword COPY followed by a list of identifiers separated by commas and terminated by a semicolon, as illustrated in DOWN procedure SLAVETRY of appendix 1.2:

```
COPY
   SUBSET, LAST, T, S, DEPTH;
```

If the body of the DOWN procedure contains any references to relatively global identifiers, these identifiers must have their scope explicitly extended by listing them in the COPY section. This exception to the normal scope rules of Pascal underlines the fact that the DOWN procedure executes in a remote processor. The translator could compile an implicit list of nonlocal references, but forcing the programmer to make the list, explicit helps document the extent of importation from the DOWN procedure's environment, and facilitates the translator's job as well. Note that the standard predefined identifiers are available to the DOWN procedure, and it is an error to include them in the COPY section (unless they have been redefined in the user's program).

Any variables listed in the COPY section reside within the slave. They are initialized with the current values of their namesakes in the master every time the DOWN procedure is invoked, just as if they were value parameters. The restriction against pointer and file types in DOWN procedure parameters also applies to the COPY variables. And as with value parameters, changes to the COPY variables in the slave do not affect their counterparts in the master. Note that the master may change the values of the copied variables from one invocation to another, though as a matter of programming practice it would be more appropriate to use value parameters for such a purpose.

DOWN procedures and CRITICAL procedures are not allowed in the COPY section. Ordinary procedures and functions are allowed, as long as they do not contain references to DOWN procedures, CRITICAL procedures, or the WAIT procedure described below. Of course the "copied" procedures need not be copied with each invocation; in the implementation, they may be linked to the code for the DOWN procedure proper. Thus procedures and functions that are of use to both master and slave do not have to be redefined within the DOWN procedure.

In the original Pascal-C report [8], relatively global constant and type identifiers could be referenced in a DOWN procedure without appearing in its COPY section. The issue was brought up again during the implementation of the translator when it became apparent that there is no good reason for this inconsistency (it is true that types and constants do not have to be copied at run time like variables, but then neither do procedures) and that this information is as useful to the translator as is the explicit listing of nonlocal variables, procedures, and functions. It was therefore agreed that the requirement should be extended to all identifiers whose scope must include the DOWN procedure. The only exception is that the type identifiers in the DOWN procedure's parameter list are implicitly included in the COPY list. This is a concession to the fact that it would seem rather odd to force a programmer to include these names in the COPY section after they have already been used in the parameter list above it.

Since the purpose of the COPY section is to make explicit any relatively global objects that are needed in the DOWN procedure - for the sake of the translator as well as the reader - it follows that any relatively global objects that are referenced in a copied procedure must also be included in the COPY list. If these objects include other procedures, the same rule applies to them, and so on until the COPY list includes all relatively global identifiers, no matter how indirectly they are referenced. Any omission causes an error to be reported. This rule is relaxed for constants and types in the COPY section, in that it is not necessary to include the named constants and types that may have been used in their definitions.

The fact that the COPY section appears after the DOWN procedure heading may lead to some confusion regarding scope rules. To clarify this, it should be remembered that the COPY section merely enables the normal scope of relatively global identifiers. The items in the COPY list are still considered relatively global to the DOWN procedure, so they may be identical to identifiers defined locally in the DOWN procedure (e.g., in the parameter list) without causing a multiple-definition error. (Of course the scope of such items would be occluded by the local definitions, but they may be needed to satisfy nonlocal references that occur in some copied procedures rather than in the DOWN procedure itself.)

Finally, note one restriction that is peculiar to this implementation. For reasons discussed in section 3.7.1, the variables in a COPY list must have been originally declared with a single type identifier, not a new type definition, on the right-hand side of the declaration. This restriction is easier to remember if one thinks of formal parameters in Pascal, which also must be declared with a single type identifier.

2.3. The CRITICAL Procedure

A CRITICAL procedure resembles an ordinary procedure except for the keyword CRITICAL in the heading, as illustrated in appendix 1.2:

```
CRITICAL PROCEDURE SOLUTION (CHOICE : SUBSET);
```

It is invoked just like an ordinary procedure.

A CRITICAL procedure can only be declared in the master part of the program, and it always executes in that environment, but it can be invoked by either the master process or a slave process. It is called CRITICAL because it can interrupt the master process (when called from a slave), but once it begins execution it cannot be interrupted by another CRITICAL activation, nor will the interrupted master process resume until the CRITICAL procedure is complete. Slaves can use a CRITICAL procedure to return results asynchronously, or even output results directly, so it provides an alternative to the VAR parameter method of returning results from a DOWN procedure. The exclusion condition ensures that the CRITICAL procedure activations will be serialized, and therefore cannot interfere with one another. When a CRITICAL procedure is called from the master, the purpose is usually to perform sensitive operations that must be protected from the asynchronous actions of CRITICAL procedures called from the slaves. Note that the exclusion condition is in effect even at the time of evaluation of the CRITICAL procedure's parameters, so no action by a slave can cause an inconsistency in the parameter evaluation. The exclusion condition persists when a CRITICAL procedure calls other procedures, and is cancelled only upon exiting the procedure that set the exclusion condition.

To make a CRITICAL procedure accessible to a DOWN procedure, the name of the CRITICAL procedure must be passed as an actual parameter in the DOWN procedure invocation. The corresponding formal declaration in the DOWN procedure's parameter list is like that of any procedural parameter except, again, for the keyword CRITICAL before PROCEDURE. In the original language proposal, CRITICAL procedures were not passed as parameters to the DOWN procedure but invoked directly as relatively global objects. The current method is more flexible and easier to implement, and perhaps more suggestive of the true semantics of a CRITICAL procedure call from a slave. (With the former method, there would have to be an exception to the Pascal-C scope rules that would otherwise require - rather confusingly - that the

CRITICAL procedure must appear in the COPY section in order to be referenced directly.)

The only legitimate parameters to a CRITICAL procedure are value parameters (excluding file and pointer types); no procedural or VAR parameters are allowed. Without VAR parameters, there is no way for the slave to obtain data from the master, which is an intentional limitation in this system. Once the slave finishes communicating its request to the master, it can continue without waiting for the CRITICAL procedure to finish or even begin its execution; the DOWN procedure may even come to completion before the CRITICAL procedure does. The activation of the CRITICAL procedure by a slave therefore constitutes another dynamically created process, independent of either its slave parent or its master grandparent.

When a slave sends a CRITICAL procedure request, the request may be queued, depending on the RTS implementation, so that the slave can proceed regardless of whether or not the master is currently able to fulfill the request. CRITICAL procedures should be regarded as having high priority with respect to the master process, so if the exclusion condition is not set the activation should take place without much delay. Between activations of CRITICAL procedures, however, the master process may be given a time slice to ensure that it is not starved out altogether.

When a DOWN procedure is called, the evaluation of its parameters takes place with the exclusion condition set, as if a CRITICAL procedure were running. This protection is provided automatically because it is forbidden to call DOWN procedures from CRITICAL procedures (or, more generally, when the exclusion condition is set), so the programmer cannot use the CRITICAL procedure method to explicitly protect the DOWN procedure's parameter evaluation. The reason for the restriction against calling DOWN procedures from CRITICAL procedures is twofold. First, it would give slaves a way of invoking DOWN procedures at their own level, which would violate the master/slave principle. Second, the master is automatically blocked if it invokes a DOWN procedure when all the slaves are busy, which can lead to the CRITICAL procedure queue filling up if the exclusion condition remains set, and possible deadlock as a result.

Like DOWN procedures, CRITICAL procedures may be nested at an arbitrary level within regular procedures and functions. Unlike DOWN procedures, however, they may also be nested within each other, and may call other CRITICAL procedures or make recursive calls without restriction.

CRITICAL procedures cannot call the WAIT procedure, for much the same reasons that they cannot call a DOWN procedure: A WAIT inside a CRITICAL procedure could block the master while in the exclusion condition and thereby deadlock the system, and slaves should not have even indirect access to a synchronizing procedure that is properly invoked only by the master process. CRITICAL procedures may nevertheless call the TERMINATE procedure.

The CRITICAL procedure is a powerful feature that requires some caution on the part of the programmer, despite its built-in safeguards. Moreover, although a DOWN procedure activation may make any number of CRITICAL procedure requests (and may return VAR parameters as well), they should be called sparingly. Otherwise, a flood of CRITICAL procedure requests could easily tie up the master and overload the communication channels, causing a drastic deterioration in performance.

2.4. The WAIT Procedure

The WAIT procedure is predefined in Pascal-C. Its invocation is illustrated near the end of appendix 1.2:

```
WAIT (SLAVETRY);
```

Note the unusual fact that it takes the name of a DOWN procedure as its parameter; in fact, a DOWN procedure name is the only legitimate parameter to the WAIT procedure.

The WAIT procedure is used to synchronize the master process with the slave processes that are executing the specified DOWN procedure. The master will pause until all activations of that DOWN procedure, and all their CRITICAL procedure requests, have come to completion. Only at that point are the actual VAR parameters connected with that DOWN procedure updated. Until then, the VAR updates returned by the slaves were kept in temporary storage.

The master then proceeds past the WAIT statement. It may create further instances of the same DOWN procedure, but these are considered a new set, and they must be concluded at some point with another call to WAIT. It is an error to come to the end of the program unless every set of DOWN procedure invocations has had a subsequent WAIT statement issued against it. The WAIT may have to be called within a more restricted region if some of the actual VAR parameters are local variables in a procedure: It is an error to attempt an update on an actual VAR parameter that has already been deallocated.

The WAIT procedure cannot be invoked, neither directly nor indirectly, by a DOWN procedure or a CRITICAL procedure. This ensures that the strict hierarchical structure of the system is not violated and prevents the possibility of deadlock.

Unfortunately, there is no way for the WAIT procedure to distinguish among the various activations pertaining to the DOWN procedure: They are all dealt with as a single group. A future version of Pascal-C may change this, provided that some reasonable method can be devised for identifying individual activations or subgroups of activations.

2.5. The TERMINATE Procedure

Like the WAIT procedure described in section 2.4, the TERMINATE procedure is predefined in Pascal-C and takes a DOWN procedure name as its sole parameter, as illustrated within CRITICAL procedure SOLUTION in appendix 1.2:

```
TERMINATE (SLAVETRY);
```

The master uses the TERMINATE feature to bring all activations of a DOWN procedure to a premature but orderly halt. The usual occasion for doing this is when enough solutions have already been returned (typically via CRITICAL procedure calls from the slaves) and no further computation is necessary. Often a single solution is all that a particular problem requires. The TERMINATE feature can thereby shorten the expected length of the computation, though of course it will not affect the worst-case behaviour.

A TERMINATE procedure may be invoked anywhere within the scope of the DOWN procedure it references, except within the DOWN procedure itself. It shares the WAIT restriction against being called by a DOWN procedure, but unlike WAIT it may be called by a CRITICAL procedure, even if the CRITICAL procedure was requested by a slave.

The TERMINATE procedure pre-empts all activations of the specified DOWN procedure, whether current or pending, and returns the slaves that were executing them to an idle state. Similarily, it cancels all CRITICAL procedure requests that have been issued by those activations. (When TERMINATE is called by a CRITICAL procedure, however, that particular activation continues through to its normal end.). The effect of TERMINATE persists afterwards until a WAIT is issued on that DOWN procedure, thereby concluding the entire set of activations. Thus if the master continues to invoke the DOWN procedure with the termination condition set, the actual parameters in the invocation will be evaluated (as discussed in section 2.1) but no activation will take place, a condition that can only be cancelled by issuing the appropriate WAIT. When a WAIT is issued with the TERMINATE condition in effect, no VAR updating takes place; the update information, if there is any at the time, is simply discarded.

It retrospect, it would have been more consistent to forbid calling TERMINATE from CRITICAL procedures, since neither WAIT nor DOWN procedures can be invoked from

CRITICAL procedures. In the latter cases, there is a danger of deadlock that does not apply to TERMINATE, but another reason for such restrictions is to prevent slaves from controlling other slaves at the same level of the hierarchy. Hence TERMINATE violates a basic premise of a hierarchical system when it is called by a CRITICAL procedure that in turn was called by a DOWN procedure. Nevertheless, this relaxation is undoubtedly an advantage from the point of view of programming convenience. The alternative would be explicit statements in the CRITICAL procedure to set some sort of termination flag, which the master process would have to test periodically, calling TERMINATE when the flag so indicates.

Like WAIT, TERMINATE refers to all activations of a DOWN procedure and cannot distinguish individuals or subgroups of activations. This capability might well prove useful (e.g., in branch-and-bound problems) and is a likely candidate for a future language extension.

3. The Pascal-C Translator

This chapter, the core of this thesis, describes the implementation of the Pascal-C translator. It is divided into sections which correspond to the major modular divisions or phases of translation. The translator itself is a program written in Pascal. Associated with it are various auxiliary pieces of software, some written especially for this project, which are described in chapter 4.

The current translator generates sequential Pascal rather than low-level code, so in this respect it is a preprocessor. This short cut was taken to get a prototype Pascal-C system in operation. In many respects, nevertheless, the translator performs much like a compiler, and indeed a good deal of it can be used in a future compiler. With this possibility in mind, some care was taken to modularize the different phases of translation to make it easier to extend the program eventually to generate low-level code.

Section 3.1 outlines some general considerations regarding error processing. Although error checking is distributed throughout the analysis phases, there is a central mechanism described in this section for logging errors. Recovery from syntactic errors is a special topic that is dealt with in section 3.4. The listing of error messages is also treated separately in section 3.3. In general, the translator will detect and report most of the errors that can be handled by a compiler.

Section 3.2 describes the lexical analyzer. This is the first phase of translation and often the most time-consuming. Attention here was given to making lexical analysis efficient and uniform in structure. To this end, the lexical analyzer recognizes several textual elements that are not considered tokens of the language: end-of-line, end-of-file, comment string, erroneous symbol, and so forth. Each kind of textual element has an action associated with it. Besides generating tokens for the parser, the lexical analyzer controls the input of the source text, the output of the listing, and the processing of translator directives embedded in the comments.

Section 3.3 describes the listing mechanism. This is fairly conventional except for the manner in which error messages are generated. Some effort was made to make error messages more meaningful than is usually the case with compiler listings, while at the same time making the generation of these messages as systematic as possible to allow for easy modification. This is particularly important with regard to syntactic errors, since syntactic analysis is now usually based on systematic methods that are difficult to integrate with an ad hoc approach to error processing.

Section 3.4 describes syntactic analysis and recovery from syntactic errors. The former is a well-understood aspect of compiler design, and this parser was based on one of the many systems now available for generating parsing tables automatically. Unfortunately, error recovery remains poorly understood by comparison. A systematic parsing method calls for a systematic recovery strategy, just as it demands a systematic approach to generating error messages. The present recovery strategy is based on various schemes suggested in the literature, in a unique but effective combination arrived at partly through experimentation.

Section 3.5 gives a broad outline of the semantic routines, what they cover and how they are linked to the parsing mechanism. This section is preliminary to the two major partitions of semantic routines described in the next two sections: bookkeeping and synthesis. The features described in all the preceding sections of this chapter are about as complete as they would be in a typical compiler. It is in the semantic routines that the translator takes advantage of the short cuts possible when the target is a high-level language. More specifically, the data declaration parts of the source program are analyzed in detail to find the exact size in machine units of all data items, since this information is vital to communicating with the RTS. Here the translator is still comparable to a compiler. But the statement parts are not analyzed in semantic detail because no low-level code is generated. Some effort is made to detect common semantic errors in the statements, and statements that invoke Pascal-C's special features are processed in more detail, but on the whole this major aspect of compiling is bypassed.

Section 3.6 describes the organization of the symbol table and the routines associated with it. The most notable aspect of this component is that a layered approach was used in designing these routines, so that the storage details of the symbol table could be hidden from the rest of the translator. An attempt was made, within the limits of the language of implementation, to deal with objects in the symbol table as abstractions that possess certain attributes and unique names (as opposed to identifiers, which are attributes that are not necessarily unique and may in fact not be possessed by some objects). This should make it easier to alter the symbol table organization without unduly disturbing the rest of the translator.

Finally, section 3.7, by far the longest in this chapter, describes the appearance and function of the modules produced by the translator, and the multi-pass scheme for producing those modules. It is divided into subsections describing the general features of the output, the particular features of the various modules, and the synthesis scheme. The key to this section is understanding how the modules produced by the translator interact with the Pascal-C run-time system. The special features of Pascal-C are translated in part as "internal" code and in part as calls to the external RTS.

3.1. Error Processing

Ideally, the translator should discover all distinct errors on the part of the user, and report only those errors. In practice, there are many kinds of errors that are difficult or impossible for the translator to detect. Some of these errors can be relegated instead to the error-checking mechanism of the run-time system. Subscript expressions, for example, are usually checked at run-time to ensure that they do not exceed the bounds of the array. Other errors may escape both the translator and the run-time checking. Still others are detected at translation time but have a disruptive effect on the translator that results in spurious or redundant error messages at various places, or conversely, in failure to report genuine additional errors. An attempt was made to minimize the disruptive effects of such errors in the Pascal-C translator (see section 3.4), though they cannot be eliminated altogether. Finally, some features in the program may provoke error messages, not because they are strictly speaking incorrect, but because they exceed certain implementation limits enforced by the translator, such as the maximum size of integer literals. Despite these limitations, error checking is an important component of translation and susceptible to varying degrees of refinement.

When a translator's output is a high-level language closely resembling the source language, as in the case of a preprocessor that converts structured Fortran to more primitive Fortran, much of the translator's error checking in the first stage may be omitted, since the subsequent compilation can be expected to discover those same errors. In this way the first-stage translator may be simplified considerably - at some inconvenience to the user, who would thereby be compelled to run two translations for every error-correction cycle, and to relate the second-stage error messages to the original source text. In the present case, on the other hand, the translator's task necessitates most of the analysis ordinarily performed by a compiler; it is mainly in the absence of low-level code generation that this translator is a simplification of a regular compiler. Thorough analysis entails thorough validation, so most errors are discovered, and reported, by the Pascal-C translator. This is of particular benefit to the user since the output of this translator is sufficiently different from the input to make it relatively difficult to relate subsequent compiler messages back to the source program. This translator reports all lexical and syntactic errors, and most semantic errors as well. The major omissions are in the statement parts, where the translator skips most of the analysis associated with low-level code generation, and therefore neglects to check for the relevant errors. In particular, although there are checks to ensure that objects used as variables, arrays, records, and so on are declared as such, there is nevertheless no validation of the various aspects of type compatibility within the statements. Since procedure calls may involve such special Pascal-C features as DOWN procedure invocations or synchronization statements, the translator does more error checking on procedure calls than on other kinds of statements.

There are two levels of error severity recognized in this implementation, and two corresponding error-logging routines. Fatal errors are those for which recovery is considered impractical. Such errors often result from exceeding critical implementation limits, such as the capacity of the symbol table. The logging routine for fatal errors displays the appropriate message directly on the user terminal, along with the line number in the source program at which the disaster occured, and then causes the translation to halt. Recoverable errors are more typical, and can be expected in some abundance. The messages for recoverable errors are incorporated into the listing, with only a summary of the errors appearing on the terminal. The logging routine for recoverable errors encodes the relevant information into an error record, which is appended to the error records already accumulated with regard to the current line. After the current line has been completely processed, the listing routine decodes the error records into textual messages and disposes of them. The information passed to the logging routine and stored in the record includes the kind of error, the position in the line at which the error was detected, and an integer value whose interpretation depends on the kind of error. This information guides the listing routine in constructing the text of error messages, although in many instances the integer item is not applicable. The logging routine keeps the list sorted using the position item as a key; that is, the records are maintained in left-to-right order of text position. Since errors may be encountered in nearly all phases of translation, calls to the two error-logging routines are very widely distributed in the translator.

3.2. Lexical Analysis

The lexical analyzer is invoked whenever the parser needs a new token. Lexical analysis operates directly on the source text, and may therefore be considered the initial phase of translation. It converts the text into a stream of tokens which the parser, in the syntactic phase of translation (see section 3.4), consumes as terminal symbols of the grammar. In some cases the lexical analyzer also associates semantic information with the token for the benefit of translation phases beyond syntactic analysis. Since it is so closely associated with the source text, the lexical analyzer is also used to control the input and listing of the source program, and to process translator directives that may be embedded in the comments.

The current line of text is stored in a character buffer, with pointers that move down the buffer as the lexical analysis proceeds. To simplify scanning, a sentinel value after the last valid character in the buffer indicates end-of-line, unless the buffer contains the last line in the file, in which case an end-of-file sentinel is used instead. When a token beyond the buffer is needed, the current line is listed (see section 3.3) and the next line, if any, is read into the buffer.

We may distinguish two subprocesses in lexical analysis: scanning and screening [9,p.109]. Scanning groups sequences of characters into distinct textual elements. Screening determines which token, if any, is represented by the textual element that was just scanned, and may supplement the token value with additional information.

Scanning can be handled by a finite-state machine, implemented either procedurally or with a transition table and driver. There are automatic generators that produce a scanning table from a regular grammar or regular expressions [10,p.124]. But scanning, unlike parsing, is a relatively simple process and hardly justifies the overhead in using automatic generators and adapting their output to suit the translator (for an illustration of this problem, see the discussion on automatic parser generation in chapter 4). The table would also occupy a substantial amount of memory and would result in slower execution as compared with the procedural method. The scanner was therefore implemented procedurally.

Screening, on the other hand, is a largely ad hoc process, and accounts for most of the code in the lexical analyzer. The procedural framework required for screening is easily extended to include scanning actions. In this implementation, a case statement selects the appropriate scanning and screening actions based on the first character of the current textual element. (Virtually all textual elements in this language can be identified upon examining their first 1 or 2 characters.) If the current textual element is recognized as a token, lexical analysis is suspended to give other phases of translation a chance to process the associated information, which is passed in global variables. Most tokens are represented only by their token values, with the following exceptions. Integer literals are accompanied by their numerical values (which may be needed for subrange calculations, for example). Character and string literals are stripped of their extraneous quotes and passed in a special buffer. Finally, identifiers are truncated to 10 characters, converted to upper case, and stored in this form in the special buffer, and are also accompanied by pointers to the corresponding entries in the symbol table (the table lookup is used as well to distinguish an identifier token from keyword tokens).

The tokens that may be produced by the screening process include not only the basic elements of the language, but also certain symbols added for convenience, such as a token to represent a character not in the lexicon (which implies a lexical error) and a token to represent end-of-file. Tokens are always passed to the other phases of translation, but other textual elements are not. If the current textual element is not a token, some action may be taken but afterwards the scanning and screening are repeated until a token is finally encountered. Non-tokens include strings of blanks, which are simply skipped, and ends-of-line, which result in listing and inputting actions. Comment strings are also non-tokens, and may result in end-of-line actions if the comment goes over line boundaries, or in the processing of translator directives if any are present.

The only translator directives that have been implemented so far are concerned with control of the listing (see section 3.3), but the processing of directives in the translator has been modularized to allow for easy additions to the repertory.

Several kinds of lexical errors are distinguished. String literals may be empty, too long (more than 80 characters) or unclosed at end-of-line. Integer literals may be too large, while real literals may have bad formats.

The input line itself may be too long, which results in truncation. A comment may still be unclosed when end-of-file is reached. All of these errors have individual messages associated with them, including positional information (in the case of an unclosed comment, the line number at which the comment apparently began is included in the message). A somewhat different case is a character that is not in the lexicon (and not within a comment). The lexical analyzer screens this as a token and exits without reporting the error. Since this special token is not an element of the language, the parser will report it as a syntactic error and invoke its error-recovery mechanism (see section 3.4), which will cause any adjoining bad characters to be skipped and resume normal processing when some reasonable input is encountered. Since such lexical errors frequently provoke syntactic errors as well, treating them as syntactic errors from the start should cut down on redundant error messages.

3.3. Listing of the Pascal-C Source Program

The translator ordinarily produces a listing of the Pascal-C source program formatted with line numbers, page headings, summary information, and possibly error messages. The quality of the listing, and of the error messages in particular, are of great practical importance to the programmer in his debugging efforts. The information should be precise, meaningful to the programmer, and fairly complete without being superfluous. Translators do not often satisfy these criteria. Hence, some attempt was made in the Pascal-C translator to improve upon the typical quality of diagnostic information. At the same time, the error reporting mechanism, especially for syntactic errors, was designed to be systematic and therefore flexible. Over-reliance on ad hoc methods for reporting errors would make development and maintenance of the translator more difficult.

As noted in section 3.2, lexical analysis processes the source text and is therefore the most suitable phase to control the listing. After all the tokens in the current line have been seen by all phases of translation, error processing for this line is complete and the listing routine is invoked. The listing routine first checks whether the suppression directive is currently active (see the corresponding translator directive below). If the listing is not being suppressed, or if there are any errors associated with this line, the listing routine proceeds to output the line and its messages. If this information is too long to fit into the current page, a new page is started before listing the line. The source line is printed with the current line number in the left margin. The error messages, if any, follow on the next few lines.

The page heading identifies the translator, its host computer, and the university, and displays the time, date, and page number. To help locate the errors in a large listing, the heading also indicates the page on which the previous message, if any, appeared. The last page of the listing features a brief summary which is printed whether or not the listing is suppressed. The summary gives the number of lines in the Pascal-C program, and either the number of errors detected or else a message indicating no errors. If there were errors, it gives the page and line numbers of the first and last errors.

The listing can be suppressed by embedding the 'L-' directive as a comment (see section 3.2 for a discussion of translator directives). The 'L+' directive resumes the listing, so that selected parts can be suppressed if desired. Whenever an error is detected by the translator, however, the offending line is listed regardless, along with the associated messages. The 'N+' directive forces a new page, while the 'N-' directive suppresses paging. The 'Ni' directive, where i is an integer literal, resets the number of lines per page from the previous (or default) value to i.

As mentioned in section 3.1, the current line may have a list of error records associated with it, one for each error detected while processing the line. The listing routine decodes this information into messages directed at the user. The rest of this section describes how these messages are generated.

The listing routine scans the error records twice. The first scan picks up the error locations stored in the records. In the listing, a line consisting of error pointers is printed just under the source line, so that the error locations are indicated graphically. The letter 'A' appears under the leftmost error, 'B' under the next, and so on. (Note: the letter points to the first character in the textual element that was being processed when the error was detected; the symbol(s) that actually need to be corrected may have occurred somewhat earlier.) The second scan picks up the other information in the error records to construct the messages below the source line and the pointer line. The letter used to point to the error also appears in the margin of the corresponding message, to make it easier to refer the messages to the appropriate locations in the source code when a large number of errors are involved. (When there are two distinct errors reported for a given location, the same letter pointer appears in both messages.) Each message appears on a separate line. The errors are numbered in sequence from 1 onwards throughout the listing, with the error number appearing in the left margin of the message.

Besides indicating the locations of the errors, the pointer line shows how much text has been skipped in recovering from syntax errors (see section 3.4). The location of the last character skipped is contained in the corresponding error record, and may be as far as the end of the line if successful recovery has not yet taken place. The listing routine underlines the entire skipped portion of text with a suitably long string of '^' characters just to the right of the error pointer. When recovery continues over line boundaries, so does this underlining process. To do this, the recovery mechanism logs a kind of pseudo-error after crossing a line boundary. When the listing routine comes across the record for this pseudo-error, it processes it like a regular syntax error, except that no letter pointer is printed and the corresponding message simply explains that recovery is continuing from a previous error. Revealing the skipped portions of code so explicitly can be helpful to the user. The peculiarities of recovery may cause some good code to be skipped, and knowing this may help to account for spurious error messages farther on. It is also useful for monitoring the performance of the recovery mechanism, allowing the developer to optimally tune that feature of the translator.

For each nonsyntactic error, there is a fairly complete message which the listing routine selects according to the error type found in the error record. Additional information in the error record may play a minor role in the message; for example, the message may contain a reference to the line number of a relevant declaration.

Syntactic error messages, in contrast, are constructed systematically according to the information in the error record. The key item of information is the shift state that the LR parser was in when the error was detected (see section 3.4). Associated with that shift state is the current grammatical symbol in the parse (that is, the one most recently shifted or reduced), and, through the state's kernel, a list of the symbols that can legitimately follow (assembled from the applicable grammatical productions). These symbols are converted into their textual representations and incorporated into the syntactic error message. The result is usually quite readable and to the point, since it is based upon the grammatical productions that define the language's syntax, productions with which the typical user is familiar. For example, the error record may identify shift state 75, which is reached (say) after processing the boolean expression in an IF statement. The error routine consults the parsing tables for this state, and finds that the symbol covered by state 75 is EXP (expression) and the symbols that can legitimately follow are THEN and BINARY OPERATOR. Using another table to map these symbols to their textual representations, the error routine writes:

"THEN" OR BINARY OPERATOR EXPECTED AFTER EXP.

3.4. Syntactic Analysis and Error Recovery

Syntactic analysis is probably the best-understood aspect of translation. A variety of efficient formal techniques are available to choose from. Most important, many products have been developed to generate a set of parsing tables automatically from a given grammar, provided that certain restrictions intrinsic to the parsing method are adhered to in the grammar. In this case, a product called SLRGO was used. SLRGO produces a set of SLR tables [10,sec.6.3] which are used in the translator by a hand-written parser driver. SLR is a member of the bottom-up LR family of parsing methods, which by most criteria are among the best available.

A practical translator should be able to recover effectively from typical errors in the source and generate fairly specific error messages. Lexical and semantic errors tend to be rather local in their effects, but syntactic errors disrupt parsing, and it is the parser that determines the structure of the program and controls all other aspects of translation. There are as yet no systematic methods of recovery that can compare with the highly developed techniques for parsing correct inputs. Given the lack of widely-accepted methods, developing an effective error strategy for the translator was an object of special interest and will be dealt with at some length in this chapter.

Recursive descent is a parsing technique that lends itself well to ad hoc error strategies. Since the parsing actions are distributed over a set of procedures, it is natural to insert code for error-handling in the various places where errors may be detected, and to suit these actions to the context of the error. This applies to the choice of messages as well as to the method of recovery. The corresponding approach for a table-driven parser is to construct a CASE statement that selects the appropriate set of actions based on the current state of the parse. This time, however, we must match the actions with the seemingly arbitrary values that represent different parsing states. Relating states to the original grammar is especially difficult if, as is usually the case, the tables are generated by someone else's program. Much of the attraction of automatic parser generators is lost if one is compelled to interpret the output in this manner, and maintenance becomes a problem.

Clearly a systematic strategy is needed for handing errors in a table-driven parser. "Panic mode" is a simple systematic method that is still much in use. Once the error has been detected and reported, the panic mode strategy is to discard input symbols until encountering one of a set of delimiting symbols; that is, symbols such as TYPE or END that serve as milestones in a Pascal program. The parsing stack is then popped until the topmost state has a parsing action for the delimiting symbol, whereupon normal parsing resumes. The optimal set of delimiting symbols is best determined by experimentation on some representative errors. If the set is too small, the parser will skip too much input and may therefore fail to detect distinct errors; if too large, the parser will not skip enough to give a reasonable assurance of success after recovery, which may lead to a series of messages provoked by a single error. A large set of delimiting symbols also makes it more likely that when a given delimiter is encountered, no state with a corresponding parsing action will be found in the stack. The panic mode algorithm should therefore include a check for such a failure, skipping to the next delimiter until a matching state is found in the stack or else the input is exhausted. This consideration is often omitted in descriptions of the panic mode method.

Panic mode works quite well in many cases but very poorly in others. Suppose, for example, that a syntax error is detected within a variable declaration. The token ";" is a likely choice for the set of delimiters, since it marks off many basic program units: variable declarations, constant definitions, statements, and so on. We would expect the recovery mechanism to skip the remainder of the faulty declaration, recover at the semicolon, and parse the subsequent variable declarations normally. If one or more variable declarations have aleady been successfully parsed before encountering the error, there will be a state near the top of the stack that represents a nonterminal symbol such as variable_declaration_list, and from that state there is indeed a parsing action for ";" (and only for ";"). If, on the other hand, the error occurs in the first variable declaration of the series, this state will not yet have been put on the stack. The stack will be popped down to the state representing VAR, which opens the variable declarations. Since ";" cannot follow VAR, the stack will be popped further until, perhaps, the state representing type_definition_list is encountered. At that point ";" is acceptable and recovery takes place, but all subsequent variable declarations in this section will be treated as erroneous type definitions. A test confirmed this drastic deterioration in performance when the error occurs in the first variable declaration.

A strategy proposed by Aho and Johnson [11] for LR parsers does not have this problem. To implement their method, certain productions must be added to the grammar. On the left-hand side of each new production is a nonterminal symbol that represents a basic unit such as a variable declaration, while on the right is a special terminal symbol, "error", which can never actually occur in the input. The parser replaces the current input token by this special symbol when it detects an error. The stack is popped until the topmost state has a parsing action for "error". The normal parsing actions that follow will reduce "error" according to whatever production is appropriate in the context. In our example, this will leave at the top of the stack the state representing variable_declaration, which will in turn be reduced to variable_declaration_list. The input symbols are then skipped until one is encountered that can follow the symbol represented by the topmost state. Since ";" is the only token that can follow variable_declaration_list, in our example recovery will take place at that point, as it does under panic mode. Unlike panic mode, however, this method ensures that the context remains that of variable declarations, since we will have variable_declaration_list at the top of the stack even if the error occured in the first declaration of the list.

The translator implements a version of this strategy, with a few modifications to alleviate certain problems. To illustrate the most serious problem, consider another example of a syntax error: a missing semicolon between the last type definition and the beginning of the variable declaration section. Using the unmodified recovery strategy just described, the parser will replace VAR with the special symbol "error" and reduce "error" to the symbol type_definition. Eventually the top state will represent the symbol type_definition_list, which can only be followed by ";". So the first variable declaration will be skipped, recovery will take place at the following semicolon, and all subsequent variable declarations will be treated as erroneous type definitions. This is the same undesirable effect produced by panic mode in our earlier example. The two situations are in fact symmetric; this method fails because it discards too much input, panic mode fails because it discards too much of the stack. By combining the two methods, however, these flaws can be cancelled out, as will be shown below. The second problem is what to do if no state with a parsing action for "error" can be found in the stack. This problem, like the first, can be solved by incorporating panic mode as a backup to the Aho and Johnson strategy. The third and final problem concerns the practice of replacing the input symbol by "error" at the time of detection. It is possible that the nature of the error is a simple omission of one or more tokens, in which case the current input symbol is not incorrect but merely premature. It may in fact be needed to recover after the error reductions, so that in discarding the current input symbol we risk discarding much of the correct text that follows. This is avoided by considering the error symbol as an insertion in front of the current input symbol, not as a replacement of it. (Note that this by itself does not solve the first problem discussed above. After the insertion, VAR will still be an unacceptable input, since type_definition list will still be at the top of the stack and VAR can only follow ";". Hence panic mode will be invoked and VAR will be discarded, leaving exactly the same situation as would result from its replacement.) The insertion of symbols by the recovery mechanism is in general dangerous, since it carries the risk of getting into an infinite loop. In this case, however, the danger is averted because the current input symbol is always consumed before recovery is completed, either as a legitimate input to the parser following the processing of the "error" symbol or as an item discarded in panic mode. As long as some input is consumed in every recovery cycle, the parsing must eventually terminate.

The algorithm for error recovery, then, is as follows. Report the error upon detection. Search the stack for the topmost state with a parsing action for "error". If one is not found, go into panic mode; otherwise, continue. Pop the stack if necessary to uncover the state, and perform the regular parsing actions considering "error" as the next symbol. One or more reductions will follow. When it is finally time for another shift, determine whether the current input symbol (which was current at the time of detection) can follow the topmost state. If it cannot, go into panic mode; otherwise, resume normal parsing. (The "improved" version of the panic mode algorithm has already been described.)

The performance of this recovery algorithm has been encouraging. Note what happens in our second example, assuming that VAR is in the delimiter set. Following the reduction of the error symbol to type_definition_list, VAR cannot follow the top state, only ";" can. So panic mode is entered, and VAR is not skipped. Instead, the stack is popped until VAR can follow the top state, in which case normal parsing resumes. Tuning efforts indicate that a small number of well-chosen error productions is sufficient, while the set of delimiting symbols is best kept large. In this combination, it is the error productions that determine the "granularity" of recovery, while the delimiting symbols play a secondary role. Only the ubiquitous identifiers and integers are excluded from the delimiting set in this implementation.

Just as a table-driven parsing technique calls for a systematic recovery mechanism, so too does it call for a systematic way of generating error messages. Ad hoc messages can be very good in some cases if they are based on experience with the most common programming errors, but systematic methods can also produce adequate messages, and by all other criteria they are superior to ad hoc methods. As is the case with recovery, both methods are limited by the difficulty of guessing what the precise nature of the error was, since that depends on the user's intention. A reasonable approach is to tell the user at what point in the parse the error was detected, and what input was expected (as opposed what was actually observed), and let the user determine the nature of the problem from that "honest" information. The symbols used in the message should be meaningful to any user who is familiar with the language definition. Here LR grammars have a distinct advantage over their LL counterparts. If one starts with a well-known formal description of the syntax (Backus-Naur form, for example), it generally requires fairly minor changes to transform this to an LR grammar. With LL grammars, in contrast, the left factoring and elimination of left recursion lead to a considerable distortion of the original productions and the addition of many new and rather arbitrary nonterminal symbols. This makes it difficult to relate parsing states to recognizable language constructs.

The following is an explanation of how a parsing state in our own SLR(1) parser is associated with the symbols used in the message.

Each shift state in any SLR grammar corresponds to an LR(0) set of items [10,sec.6.2]. The kernel of the set [10,p.236] comprises the most important items, the highest-level productions associated with the state, together with a pointer that indicates how many symbols on the right-hand side of the productions have already been processed. The symbols to the left of the pointer are necessarily the same for all productions in the kernel, while those to the right may differ depending on what different productions may be consistent with what has already been seen. The message is constructed by listing all the symbols that are expected at this point, that is, all the symbols to the immediate right of the pointer, followed by "expected after", followed by the symbol most recently parsed, the one to the immediate left of the pointer. Note that the symbols listed may be nonterminals as well as tokens; in fact, they will be the highest-level grammar symbols in that context. To return to the example presented in section 3.3, shift state 75 has a kernel consisting of two items, with a dot representing the pointer:

```
IF STAT  ->  "IF"   EXP . "THEN"   STATEMENT
EXP      ->  EXP . BINARY OPERATOR   SIMPLE EXP
```

The corresponding message is:

```
"THEN" OR BINARY OPERATOR EXPECTED AFTER EXP.
```

Note how the kernel gives the succinct nonterminal symbol BINARY OPERATOR so that one is not compelled to list all the corresponding terminal symbols ("+" OR "-" OR "*" etc.). Clearly, the quality of the messages depends largely on the choice of meaningful, familiar symbol names in the grammar, since the same grammar that is input to the parser generator is used to derive the text for these messages (see chapter 4).

3.5. Semantic Routines

The semantics of a language typically account for most of the code in a compiler, and this is indeed the case with the Pascal-C translator. The semantic routines perform a variety of tasks which can be grouped under two basic headings. First, they complete the analysis begun in the lexical and syntactic phases, particularly with regard to collecting information on user-defined symbols and checking them for consistent use. These are aspects of analysis that cannot be conveniently handled in the earlier phases. Second, the semantic routines generate the target code. The present section describes the semantic routines generally, and how they are linked with the rest of the translator. The two subsequent sections elaborate on the semantic routines: Section 3.6 explains how symbol information is maintained and section 3.7 describes the generation of the target modules.

The declarative portions of the source program are analyzed relatively thoroughly in this translator, much as they would be in a compiler. In the executable portions, on the other hand, the semantic analysis is much simpler than it would be in a compiler. Part of the reason for this difference is that the Pascal-C declarations, particularly of DOWN procedures, require more complicated changes than any statements in order to translate them into sequential Pascal. More significantly, the translator is required to extract low-level information from data declarations but, since the target code is a high-level language, not from statements. The low-level information required from data declarations is the exact size in bytes of any variable that may be passed from one processor to another. This passing is done via RTS routines (see section 3.7), and the size of the item being passed is one of the essential parameters in such RTS calls. The only kinds of statements that do require special attention by this translator are procedure calls, since they may involve specific Pascal-C features, and WITH statements, since field references contained in the WITH statement must be properly resolved.

The extent of semantic error checking reflects the depth of semantic analysis required in different sections of the program. Thus the error checking for the declarations is much more complete than for the statements. In statements that contain expressions, in particular, there is none of the usual validation of type compatibility among operands and operators. Nevertheless, the level of analysis within statements is sufficient, with a little additional effort, to allow detection of most of the common errors. As explained in section 3.1, the translator detects and reports errors wherever it is reasonable to do so, even if similar error checking will be subsequently performed by the Pascal compiler on the translator's output.

The translation is syntax-directed; that is, all the semantic actions are linked to productions in the grammar, which in a bottom-up parser are in turn associated with parser reduce states [10,p.246]. Whenever the parser performs a reduction, it activates the semantic phase of translation through a semantic control procedure. A CASE statement in the control procedure selects whatever semantic actions, if any, may be appropriate for the current reduce state. To make this part of the translator more readable, each CASE element is accompanied by a comment that shows the corresponding grammatical production. (See chapter 4 for an account of how these comments are automatically generated.) The statements in a CASE element consist mostly of calls to one or more semantic subroutines. For reduce states that do not require any semantic actions, the corresponding CASE elements are simply omitted and control falls through to the empty OTHERWISE clause.

Closely associated with the semantic control procedure is the semantic stack, a data structure which is effectively an extension of the parsing stack. Its purpose is to temporarily store semantic information in parallel with the syntactic information contained in the parsing stack. The semantic control procedure is exclusively responsible for maintaining the semantic stack, just as the parser "owns" the parsing stack, but both stacks are implemented as global structures since in Pascal we cannot declare statically allocated variables as local to a procedure. In a reduction, the parser removes the top elements from the parsing stack, which correspond to the symbols on the right side of the grammatical production in question, and replaces them with a new element, corresponding to the symbol on the left side of the production. The semantic control routine echoes these moves, but before discarding the top elements of the semantic stack, it may pass the information contained therein to the semantic routines, and these may return some information to be incorporated into the new element at the top of the stack.

3.6. Bookkeeping

The role of the symbol table is to keep track of all identifiers used in the source program and their attributes, as well as predefined identifiers, keywords, and identifiers that are referenced but as yet undefined. Labels and literals are not preserved in the symbol table in this implementation. Labels, in fact, are virtually ignored beyond the syntactic phase, but literals are evaluated and, if the literal appears within a constant or type definition, its value may be stored in the symbol table as part of the information for that constant or type.

The bookkeeping routines perform storage, retrieval, and deletion operations on the symbol table. Since symbol information is so ubiquitously referenced, the bookkeeping routines fulfill the important function of hiding the storage details from the rest of the translator. The symbol table together with its bookeping operations would be defined as a module in a language like Modula [12], but with Pascal as the language of implementation we are compelled to make the symbol table a global data structure. The translator follows an informal discipline, avoiding any reference to the symbol table except via a clearly identified set of bookkeeping routines.

The translator's initialization routine performs the first bookkeeping operations: the insertion of keywords and predefined identifiers in the symbol table. Subsequently, as declarations (or undefined references) are encountered in the source program, new entries are inserted. The information in a given entry may be extended or updated as the declaration is processed, or in some cases later on in the translation. At the end of a subprogram declaration, the corresponding block of local objects is popped off the symbol table, except for the information (e.g., parameter list) needed to check subsequent calls to the subprogram.

The majority of bookkeeping operations do not perform insertions or deletions but access the individual attributes of existing objects. A typical attribute has a pair of associated access routines, one for storage (using a value parameter) and one for retrieval (using a function value). The object in question is identified by its key, a unique value generated by the bookkeeping routine that inserts the new object, and of significance only within the translator. Henceforth an object's "key" will be understood to mean this value, while the character string that denotes the object in the source program will be called its "identifier" to avoid confusion. There are many reasons why, for internal purposes, the key is a more convenient way to denote the object. In the first place, there is a one-to-one correspondence between existing objects and valid keys. An object's identifier, in contrast, may be occluded within an inner block because another object is defined there with the same identifier. Moreover, some objects may be defined anonymously, which is to say without identifiers: variants within records, or structured types within variable declarations, for example. Another important consideration is that keys can be implemented as compact, scalar values, and are therefore much more efficient as object pointers in terms of both computation and storage space.

When identifiers are encountered in the source text they are converted into the equivalent keys. The bookkeeping routine that performs this mapping is unique in that it is invoked by the lexical analyzer rather than by the semantic routines. (Recall from section 3.2 that the lexical analyzer, as part of its screening function, must distinguish keywords from true identifiers. This requires access to the object in question.) Once the key becomes available, it denotes the object and the identifier may be considered just another of the object's attributes.

The most fundamental attribute, which largely determines what other attributes may be relevant to the object, is its class. Keywords form one class of object; so do constants, variables, functions, and so on. Pascal types are so heterogeneous that thay are subdivided into separate classes for arrays, records, pointers, ordinals, etc. There are also a few classes that are used by the translator for special purposes, such as when an object is undefined or multiply defined (so that redundant error messages can be avoided).

Another attribute common to all objects is the defining region. For most objects, the defining region is represented by the key of the immediately surrounding block (program or subroutine), while for fields it is the key of the surrounding record. Keywords and predefined objects are considered to be defined within a larger region, one that encloses the entire source program.

In this implementation, most user-defined objects have another pair of attributes: the coordinates (line number and character position within the line) for the beginning and end of the object's declaration in the source text. Section 3.7 explains the main application of this information in synthesizing the target code. The coordinates are also incidentally used to enhance certain semantic error messages; for example, by listing the line number of the relevant declaration when an object is improperly referenced.

We shall now describe the representation of the most important classes of objects.

Constants can be defined directly in a constant definition section, or indirectly as the elements of an enumerated type. Either way, constants have a type attribute: the key of the constant's base type, which must be either a predefined or an enumerated type. The constants of greatest interest in this translator are those that have an ordinal base type (i.e., integer, boolean, char, or enumerated), since such constants may be subsequently used within the definition of a type object. For example, any ordinal constant may be a lower or upper limit of a subrange, and the subrange may in turn define the index of an array. Ordinal constants thus help determine the size of data objects, which is essential information in this translator. Hence all ordinal constants in the symbol table (which excludes literals) have a value attribute. This is an integer value, which in the case of a noninteger constant represents its ordinality. Three constants, all ordinals, are predefined: maxint, false, and true.

Types, as explained previously, are subdivided into different classes. There are the predefined types REAL and STRING, each of which is in a class by itself and does not possess any particular attributes. (Literal strings are considered to be of type CHAR if their length is exactly one, and of type STRING otherwise.) Many of the other types have an attribute that is simply a pointer to a related type object: Those of class FILE have a component type, those of class SET an element type, and those of class POINTER a domain type. A special class arises when one type is defined as synonymous with another; for example, "t2 = t1". Type t2 is of class SYNONYM and has an attribute pointing to t1 or, if t1 is itself a synonym, to the type pointed to by t1. Array objects have two type attributes: one for the component type and another for the index type, which always exists as a separate ordinal object. The class ORDINAL, as the term implies, comprises the predefined types integer, boolean, and char, the enumerated types, and all subranges thereof. All ordinal objects are treated as subranges in that their attributes include the low and high limits of the range (expressed as integers), and the key of the host type. For ordinal objects that are not true subranges, the values are those of the unrestricted range and the host type is the object itself. (This uniformity makes certain bookkeeping operations simpler to perform.) Type objects of class RECORD have two integer attributes: the number of fields in the fixed part and the number of variants. Fields resemble ordinary variables, except that in their region attribute they point to the enclosing record or variant rather than to the subprogram. The field identifiers are rendered inactive after processing the record declaration, and are only reactivated when the record is referenced in a WITH statement or in a field designator. Variants form another class with the same structure as records, since variants may themselves contain a fixed and a variant part. As with fields, the defining region of variants is the enclosing record or variant. (Note however that any constants or types defined implicitly in field declarations have the surrounding block as their defining region, just as if they were declared outide the record definition.) The tag field, if any, is simply treated as the last field in the fixed part. This translator considers the overlaying of variants in determining the correct size of record types, but is not concerned with the details of discriminated type unions. Finally, there is a temporary class for types that are referenced as the domains of pointer types but have not yet been defined, and one for types that are still in the process of being defined; i.e., in an ongoing type definition. In the absence of errors, both of these are eventually resolved to a specific class of type object. (Unresolved domain types are trapped as errors at the end of the declaration part of the block.)

Variables have a type attribute, through which their size can be determined. (Note that only the variable's size is important in this implementation; its relative starting address is not.) Value parameters are in the same class as variables, while VAR parameters and fields have their own classes but an identical attribute structure.

There are four classes of user-defined subprograms: procedures, functions, DOWN procedures, and CRITICAL procedures. The predefined Pascal-C procedures WAIT and TERMINATE have their own distinct classes, while the remaining predefined subprograms are standard procedures and functions. When a subprogram declaration is being processed, its key represents the current defining region. At the end of the declaration, the subprogram's local objects are popped from the table and the key of its own defining region is restored as the current defining region. The identifiers of the subprogram's parameters are deactivated, but the parameters are preserved in the symbol table to check the calling sequence in subsequent invocations of the subprogram. They are finally deleted when the subprogram itself is deleted. If the subprogram is declared with a forward directive, it once again becomes the current region when the forward block is encountered, and the parameter identifiers are reactivated. All user-defined subprograms have an associated block object that contains general information and also marks the end of the parameter list. One of the attributes of a subprogram object is the key of its block object. The class of the block object determines whether a real block follows or the subprogram is forward, external, or a formal parameter. If a real block exists, the block object has an attribute pointing to the first local object, if any. The subprogram object also has a type attribute if it is a function. If it is DOWN procedure or a formal CRITICAL procedure parameter in a DOWN procedure heading, it has an integer attribute: a sequence number that is used to identify the procedure in calls to the run-time system (see section 3.7).

In the current implementation, the main part of the symbol table is an array of records. Each record represents an object, and the record's index in the array is equivalent to the object's key. The fields within an object's record correspond to its attributes. (This fixed-storage scheme could be improved upon in the interest of minimizing space, but with some loss of simplicity.) There is also a hashing table to convert identifiers to keys through a hashing function. Objects whose identifiers hash to the same value are linked together in the main table, so a linear search may be required after hashing. Note that an object's identifier can be rendered inactive by temporarily removing the corresponding record from its hashing list, restoring it later if need be. (Field identifiers, for example, should be active only at certain times, although the fields are always accessible for internal purposes through their keys.) Since access to the symbol table is restricted to the bookkeeping routines, such implementation details should not be too difficult to modify in the future.

The COPY list in a Pascal-C program (see section 2.2) is represented apart from the symbol table, but is closely associated with it nonetheless. The COPY list is maintained as a linked list of keys which includes not only those objects mentioned explicitly in the COPY section, but also the types of the data parameters in the DOWN procedure heading, as well as all types and constants which are needed to close the definitions of all preceding objects in the list. These additional keys are added to the list automatically for the user's convenience, except in the case of subprograms in the COPY list. For all such subprograms, the user must ensure that any nonlocal objects referenced within them are explicitly in the COPY list or will be included by the translator when it resolves other definitions. The COPY list is retained until the end of the DOWN procedure declaration, and is then discarded. The translator uses it mainly to trap nonlocal references that ought to have corresponding entries in the COPY list. If this error is committed indirectly by a copied subprogram, however, it will escape the translator's notice and will eventually provoke an "undefined reference" error when the output module corresponding to the DOWN procedure is compiled.

3.7. Synthesis

The first part of this section gives a general description of the translator's output, focusing on those features that are common to the output modules. Following this are detailed descriptions of the various output modules. At the end of this section is an account of the translator's strategy for producing the output. To see examples of the translator's output, see the appendices, which include translated versions of the Pascal-C programs for the knapsack and quicksort problems.

3.7.1. The translator's output

The translator converts the Pascal-C source program into a set of modules in sequential Pascal. These are to be compiled and linked so that we end up with two executable programs: one to run on the master processor and the other to run on each of the slaves. The master and slave programs communicate via their repective run-time systems. One of the principal responsibilities of the translator is to insert explicit RTS calls wherever these are needed in the code, and to back these up by inserting appropriate declarations beforehand.

One of the modules produced by the translator serves as the master program, while the others are linked together to form the slave program. The master module is based on all parts of the source program that lie outside the DOWN procedures. On the slave side, there is a small main program with a number of external procedures, each one based on one of the DOWN procedures declared in the source program. The slave's main program serves to control the execution of these procedures in response to DOWN procedure requests from the master program.

The slave program is therefore capable of executing any of the DOWN procedures. If many DOWN procedures are declared in the source program, the external procedures produced by the translator should be compiled separately and linked with the slave's main program as overlaid segments, so that only the currently active DOWN procedure will be in the slave's memory at any given time. Since a typical DOWN procedure activation is expected to run for a very long time, the overhead involved in switching segments should be relatively insignificant. Even so, this overhead can be minimized if the master RTS, when it must choose among several idle slaves, gives preference to those that already possess the appropriate segment in memory (i.e., those that have executed an instance of the same DOWN procedure last time). This strategy would also be effective in a future system that did not provide disk access for every slave. In such a system, the master would have to transmit the DOWN procedure code to the slave directly, but could avoid repeating this action uneccessarily by giving preference to slaves that still have that information in memory.

The run-time system required to support Pascal-C can be divided into two parts: the conventional run-time support required for sequential Pascal, and the additional run-time support required for Pascal-C's special features. One of the advantages of generating Pascal code with this translator is that the conventional RTS is already provided with the system used to compile the translator's output modules. The special RTS needed for Pascal-C, developed by Yin-Lam Wong [5], is nevertheless very considerable, since it must perform the many hidden bookkeeping, communication, and synchronizati[illegible] tasks that support Pascal-C's powerful features. Some par[illegible] the Pascal-C RTS had to be written in assembler language for PDP-11 computers, but most of it was written in Parallel Pascal [13], a high-level language for concurrent programming that borrows its main features from Modula [12]. The fact that Parallel Pascal's sequential subset is very close to standard Pascal makes it particularly suitable for this project, since it means that the translator's output modules, too, can be compiled as Parallel Pascal modules. It is then easy to link the compiled modules with the compiled run-time routines, since the calling protocols will be compatible. Hence the entire system relies on Parallel Pascal's conventional support, and, in the case of the Pascal-C RTS, its concurrent support as well. (But see chapter 5 for a discussion of the shortcomings of Parallel Pascal's run-time support for concurrency, and the measures taken to remedy them.)

Although Parallel Pascal's sequential subset is very close to standard Pascal, there are a few minor deviations. Since the translator's output modules are to be compiled using the Parallel Pascal system, the user must ensure that the conventional Pascal features of the Pascal-C source program are compatible with Parallel Pascal. Most of the deviations from standard Pascal are in fact extensions which the user may choose to ignore. Nevertheless, some of these extensions have proven to be quite useful for the translator. When the translator must set up a declaration of its own making, for example, it may insert a new CONST, TYPE, or VAR section as needed since these data sections can be repeated within a block in any order, as long as all the local data declarations precede the local procedure declarations.

A typical parameter list for an RTS procedure includes an action code to specify the particular action requested, plus one or more other arguments to supply relevant information for performing that action. The translator must insert explicit RTS calls where needed, along with explicit declarations of those RTS procedures as externals. Pascal's strong type checking causes a difficulty here, in that sometimes the arguments of the RTS procedures are data items whose types may be user-defined and of arbitrary complexity (for example, the parameters of a DOWN procedure). It is impossible to prepare a fixed number of RTS procedures to handle every conceivable data type. Some RTS procedures must serve a variety of data types, and it is up to the translator to redefine them appropriately without provoking compiler errors. The loophole used in this case is that Pascal allows external procedures to be defined as local to ordinary procedures, yet ensures that all external procedures with the same identifier will be linked to the same external entry point. Hence an RTS procedure can be redefined as often as necessary, without provoking an error, by making the redefinitions local to dummy procedures created for that purpose. The dummy procedures, of course, all have distinct identifiers, and each one's parameter list reflects that of the redefined RTS procedure within it. The program invokes the dummy procedure instead of calling the

RTS procedure directly. The dummy procedure, in turn, calls its local RTS procedure and thereby passes on its arguments. For example, supose that in the source program we have a DOWN procedure with two value parameters, one of type T1 and the other of type T2. There is a universal RTS procedure for sending value parameters from master to slave, and it must be redefined to suit the two data types in question. The translator sets up dummy procedures for this purpose, as shown in the following:

```
PROCEDURE ZZPRO1 (ZZACTION : INTEGER; VAR ZZDATA : T1;
                  ZZSIZE : INTEGER);
  PROCEDURE RTS3 (ZZACTION : INTEGER; VAR ZZDATA : T1;
                  ZZSIZE : INTEGER); EXTERNAL;
  BEGIN
    RTS3 (ZZACTION, ZZDATA, ZZSIZE)
  END; (* ZZPRO1 *)

PROCEDURE ZZPRO2 (ZZACTION : INTEGER; VAR ZZDATA : T2;
                  ZZSIZE : INTEGER);
  PROCEDURE RTS3 (ZZACTION : INTEGER; VAR ZZDATA : T2;
                  ZZSIZE : INTEGER); EXTERNAL;
  BEGIN
    RTS3 (ZZACTION, ZZDATA, ZZSIZE)
  END; (* ZZPRO2 *)
```

Note that the dummy procedures ZZPRO1 and ZZPRO2 have distinct identifiers while the nested external procedure RTS3 appears in both, and that the type identifier of the second parameter is T1 in the first case and T2 in the second. Similar redefining takes place in other situations, on the slave side as well as on the master side. This scheme seems rather awkward and requires some machine-level knowledge of calling protocols to ensure that the RTS procedure will correctly handle those arguments that can be based on a variety of data types. Nevertheless, it is a breach of security that need not concern the Pascal-C user, but only the implementers of the system, who have the opportunity to consider and verify all of the possibilities beforehand.

Note also the use of the prefix "ZZ" in many places. It is the practice of the translator to use this prefix for all identifiers that it generates for its own purposes. Pascal-C users are expected to be aware of this fact in order to avoid conflicts with their own identifiers in the source program. Users should also beware of choosing identifiers that are prefixed with "RTS".

Besides its strong type checking, Pascal presents another problem for the translator in constructing RTS declarations. In Pascal procedure declarations, formal parameters must be defined using single type identifiers; no new type constructions are allowed within parameter lists. This is not a problem in the example cited above, but only because DOWN procedure parameters are already subject to this restriction in the source program. A variable listed in the COPY section, on the other hand, may have been declared using a new type construction. If so, the translator will not have a type identifier to use when it generates the dummy procedure and RTS redefinition for transmitting this variable. It would be possible for the translator to create an appropriate type declaration, but Pascal would then require that the new type identifier be used in the original variable declaration as well as in the formal parameter declaration. This would greatly complicate the translator's strategy for synthesizing the output modules (see section 3.7.5). Therefore, a programming restriction was added to this implementation that was not present in the Pascal-C language definition: Variables listed in the COPY section, like parameters in a procedure, must have been declared with a single type identifier. This restriction has already been mentioned in section 2.2.

Finally, before proceeding to the detailed descriptions of the translator's output modules, note that numerous comments are inserted in the text to identify the modules and explain many specific insertions of declarations or statements. These comments may prove helpful should the user find it necessary to examine the translator's output before compiling it.

3.7.2. The master module

The master module is based on those parts of the source program that lie outside the DOWN procedures. Appendices 1.3 and 2.3 show examples of the master program. The major modifications to the original code are as follows:

(a) The translator inserts the following declaration at the very beginning of the master module's global declarations:

```
VAR ZZRTSBUF : ARRAY [1..1500] OF INTEGER;
```

The sole purpose of this variable is to provide some workspace for the RTS; it is not referenced elsewhere in the master module. The RTS locates this area by referencing the starting location for the program's global memory. Note that the variable appears within its own variable-declaration section. This illustrates the convenience of being able to disregard the number and order of data-declaration sections in Parallel Pascal's syntax (as discussed in section 3.7.1).

(b) The bodies of the DOWN procedures are transferred to the slave modules, but the master retains the headings in modified form and with new bodies appended to them. The function of the modified procedure is to set up an activation of the DOWN procedure in a slave, and pass it the parameters and COPY variables.

To translate the heading of the former DOWN procedure to sequential Pascal, the keyword DOWN is eliminated, as is the keyword CRITICAL in any CRITICAL procedure parameters. The declaration part of the new body contains external declarations of the RTS procedures that are used to pass information to the slave. To adapt these RTS declarations to the appropriate data types, they are redefined within dummy procedures as described in section 3.7.1. The statement part of the new body consists mostly of calls to RTS procedures, either directly or indirectly through calls to the dummy procedures.

The first statement is an RTS call to check whether a TERMINATE statement has been issued against this DOWN procedure; if [illegible] the remaining statements are skipped. This is the logical point at which to ask this question, since Pascal-C requires that a DOWN procedure's parameters be evaluated (and any attendant side effects produced) whether or not the termination condition is in effect. The next RTS call, if executed, starts the activation by informing the RTS of the DOWN procedure identification and the number of VAR parameters to expect. Next is a sequence of calls to the dummy procedures to send down the COPY variables and the DOWN procedure parameters. When data objects are being sent, the RTS parameters in each call include the action code, the data object (as a VAR parameter), and the size of the object in bytes. There are separate action codes for value and VAR parameters. For procedure parameters (which were originally CRITICAL procedure parameters), the RTS parameters are the action code and the procedure identifier, which has the effect of transmitting the procedure's static link and entry point. The final statement in the master's version of the DOWN procedure is an RTS call that signals the end of the activation process.

(d) The statements that invoke the DOWN procedure are retained in the master module, since the headings have effectively remained the same, but the invocations are bracketed by RTS calls, two before and one after, and this whole group of procedure calls is further bracketed by BEGIN and END. The first RTS call reserves a free slave processor. The second sets the exclusion condition that blocks any asynchronous execution of CRITICAL procedures requested by the slaves. The final RTS call, which follows the DOWN procedure call, cancels the exclusion condition. Pascal-C requires that the evaluation of a DOWN procedure's parameters be protected from asynchronous events initiated by the slaves, which is why the exclusion condition must be set before the invocation of the DOWN procedure. But before setting the exclusion condition, one must be certain that a slave processor is available. To wait for a free slave while the exclusion condition is set invites deadlock. (The slaves may all be held up trying to send CRITICAL procedure requests to the master, which cannot take their requests because the CRITICAL procedure queue is full, and nothing from the queue can be consumed so long as CRITICAL procedures are excluded from execution.)

(d) There is a kind of run-time error peculiar to Pascal-C that is quite difficult to detect, though it may have disastrous consequences. This error concerns the actual VAR parameters to a DOWN procedure, which by Pascal-C's semantics are not updated until the master issues a synchronizing WAIT statement on that DOWN procedure and all activations have come to completion (see section 2.1). The problem is that by the time the WAIT statement is invoked, some of those actual parameters may have been already deallocated. The updating mechanism would then blindly overwrite whatever currently occupied those locations in memory, with utterly unpredictable results. Such a dangerous possibility cannot be permitted in the system, yet there seems to be no practical way to detect the deallocation at the time of updating.

Static error checking has only limited usefulness in this regard. Whether or not a particular combination of DOWN procedure invocation and WAIT statement actually occurs will in general depend on the flow of execution. On the other hand, the translator could enforce a stricter language provision, such as allowing only global variables as actual VAR parameters to a DOWN procedure, but such restrictions were rejected as too severe. The only alternative is to detect the error at run time.

There are two ways in which variables can be deallocated during the lifetime of a program. One way is to explicitly DISPOSE dynamic (i.e., heap) variables. This presents no problem because there is a current language restriction against passing any part of a dynamic variable as an actual VAR parameter to a DOWN procedure (see section 2.1). The RTS has access to the heap pointer, and by comparing it with the addresses of incoming VAR parameters the RTS can determine whether any VAR parameters are dynamic variables, and if so reports a run-time error.

The other kind of deallocation occurs implicitly at the end of procedures. When a procedure's block is deactivated, all local variables and formal parameters are popped from the run-time stack. If this includes some variable (or, equivalently, a formal value parameter) that has yet to be updated as a VAR parameter to a DOWN procedure, an error has occurred. This is the first point at which the error can be detected, and is perhaps the only practical point. For if the RTS postpones this error check until, say, the time comes for updating the actual VAR parameters, the stack level may by then have risen again through other procedure calls, leaving no evidence of the accidental deallocation. In this implementation, therefore, the system must monitor the stack level and trap the error as soon as the stack shrinks below the highest variable on the stack that is awaiting updating. Note that the definition of error implied by this method may be a little more severe than necessary: There may be a call later on to TERMINATE the DOWN procedure and thereby cancel the updating (see section 2.5), in which case it is debatable whether a premature deallocation should really be considered an error.

The RTS can compare the current stack level against the level of the highest actual parameter currently on the stack; it has access to or is able to maintain both these items of information. The role of the translator is to determine at which points such a check is appropriate, and insert RTS calls accordingly. The obvious place is either at the end of a procedure where an actual VAR parameter is locally declared, or just after such a procedure is called. The latter position is problematical because the procedure in question may be a function that is called in the middle of an expression. (It may be convenient to call a DOWN procedure from within a function, although technically that would constitute a side effect.) At the end of the procedure, on the other hand, the variables are still active and the RTS cannot determine in advance how much memory will be deallocated. To supply this information, therefore, the RTS call at the end of the procedure passes the first local variable as a VAR parameter, effectively giving the RTS the starting address of the current activation block. (Variables in this system are allocated memory in the exact order of their declaration. This would not be a safe assumption generally.) Since the RTS procedure includes a user-defined variable in its parameter list, its declaration is inserted as local to the procedure to allow for the redefinition of parameter types.

The remaining problem for the translator is to identify the procedures in which these RTS calls are to be inserted.

The current criterion is simple but too strong: all procedures in the master module except those with no local variables and those defined within CRITICAL procedures (it is an error to call DOWN procedures in the CRITICAL state, as explained in section 2.3). A more refined criterion could be implemented in the future through the use of data-flow analysis to trace back actual VAR parameters from the DOWN procedure calls to the procedures in which they are declared as local variables; only those procedures are at risk for accidental deallocation.

(e) CRITICAL procedure declarations do not require any modification beyond eliminating the keyword CRITICAL. Every call to a CRITICAL procedure in the master (calls from the slaves will be considered in section 3.7.3) is preceded by an RTS call to set the exclusion condition and followed by an RTS call to cancel the condition, all bracketed by BEGIN and END. This ensures that the scope of protection for the CRITICAL procedure includes the evaluation of its parameters, as required by Pascal-C's specifications (see section 2.3). Note that CRITICAL procedures may be recursive or call each other, so the RTS must keep track and cancel the exclusion condition only after the first CRITICAL procedure activation in the series has ended.

(f) Calls to the Pascal-C synchronization procedures WAIT and TERMINATE are replaced by the equivalent RTS calls, with the original calls retained as comments for legibility. The first parameter to the RTS call, the action code, distinguishes between WAIT and TERMINATE. The second parameter identifies the DOWN procedure in question.

(g) The master module ends with the main statement part, which corresponds to the same part in the original Pascal-C program. The translator inserts an RTS call as the first statement. This informs the RTS of the number of DOWN procedures declared in the program and tells it to perform its initialization tasks. Another RTS call is inserted as the last statement, this time to trigger the final housekeeping chores.

3.7.3. The main module for the slaves

The main slave module is a small program generated by the translator (see appendices 1.4 and 2.4 for complete examples). It is to be linked with the DOWN procedures, which are issued separately as external procedure modules (see section 3.7.4). The core of the main slave program is a loop to obtain DOWN procedure requests from the master and activate the corresponding external procedures, as follows:

```
REPEAT
 RTS6(160,DOWNID); (* WAIT FOR DOWN PROC REQUEST *)
 CASE DOWNID OF
  1:DP1;
  2:DP2;
  etc.
  OTHERWISE
 END;
UNTIL DOWNID = -1
```

Procedure RTS6 returns the down procedure identification in parameter DOWNID after receiving this information from the master, which may involve an indefinitely long wait. DOWNID is of type integer. DP1, DP2, etc. are the names given to the external procedures by the translator, in their order of occurence as DOWN procedure declarations in the source program. At the end of the program the master sends a DOWNID value of -1 instead of a legitimate value to let the slave programs exit in an orderly fashion.

There are two initialization statements in the main slave module that precede this loop. The first is an RTS call to perform the housekeeping chores. The second is a call to a low-level procedure known as ZZSTAT which has been incorporated into the Parallel Pascal RTS (see chapter 5). This procedure saves important information that defines the current status of the slave program (program counter, stack pointer, and so on) in memory locations known to the Pascal-C RTS. This helps the RTS to return the slave program to the beginning of the loop when a TERMINATE directive received from the master causes the interruption of a DOWN procedure activation.

Note that the slave's main program, like the master program, has a special data declaration inserted at the beginning to provide the RTS with some workspace.

3.7.4. The DOWN procedure modules for the slaves

The external slave procedures are derived from the blocks of the DOWN procedures in the source program. They are collected in a single output file. (See appendices 1.5 and 2.5 for complete examples of the external procedures.) As explained in section 3.7.1, these modules are to be linked with the slave's main module (possibly as overlaid segments) after compiling all of them as Parallel Pascal modules. The translator names the slave procedures DP1, DP2, and so forth, reflecting the order of appearance of the corresponding DOWN procedure declarations in the source program. Unlike the original DOWN procedures, these slave procedures have no parameter lists. Instead, both the COPY variables and the DOWN procedure parameters appear as locally declared variables that are received from the master by issuing appropriate RTS calls at an early point in the slave procedure. For every item ~~that~~ the master sends during the activation of the DOWN procedure (recall section 3.7.2), there is a corresponding call in the slave to receive it in a local variable.

So the translator must insert local declarations for what were formerly DOWN procedure parameters and COPY variables. In addition, the translator must insert declarations for the other objects in the COPY list: constants, types, and procedures. The slave procedure is a separate module, which means that any nonlocal objects would be undefined unless they were redeclared locally. There is a difficulty in making all these insertions in that the original items may have come from different defining regions within the source program. For example, a procedure declaration may appear in the COPY list along with a type declaration from a more closely nested region farther down the source program. The translator cannot "flatten out" these two declarations within a single region and at the same time preserve their original order of appearance, since type declarations cannot follow procedure declarations in Pascal (nor in Parallel Pascal). On the other hand, inserting the declarations in a different order would make it difficult to construct the output modules in a single additional pass according to the scheme outlined in section 3.7.5. Instead, the translator inserts the declarations in their original order of appearance, but echoes the original nesting pattern whenever necessary. The declarations are laid out linearly except when a data declaration follows a procedure declaration, which requires that the data declaration be nested at a deeper level. At that point, the translator sets up the heading for a new procedure in the slave module, local to the previous procedure. The deepest level of nesting contains the local declarations for the DOWN procedure parameters, and the DOWN procedure body itself. (As explained in section 2.2, all objects in the COPY list are considered relatively global to the DOWN procedure, so the latter always occupies the deepest level of nesting.) Of course, this pattern of nesting must be unwound at the end of the module. Each level has a statement part with RTS calls to receive values for its local variables, if any, followed by a call to activate the next (deeper) procedure level.

In most cases, the simplest approach to copying a declaration is to read the original text from the source program and copy it directly to the output module. Constants, types, and procedures listed in the COPY section are best handled in this manner. For procedure declarations, in fact, there is really no alternative. But variable declarations are easily reconstructed from the symbol table, provided that a single type identifier was used to denote the variable type in the declaration. This is indeed the case for DOWN procedure parameters as well as for variables listed in the COPY section (recall section 3.7.1). Moreover, copying straight from the text could be particularly awkward for a COPY variable, since it might be declared as part of an identifier list that includes unwanted variables, in which case the declaration should not be copied wholesale.

The RTS procedures used to receive the former COPY variables and DOWN parameters must deal with arbitrary data types, and hence must be redefined in the manner described in section 3.7.1. These RTS redefinitions accompany their corresponding variable declarations at the same nesting level of the slave procedure.

For former VAR parameters, the slave procedure requires more than the inserted declarations described above. Each local variable declaration for a VAR parameter must be accompanied by a buffer variable of the same data type. The buffer is used to retain the original contents of the VAR parameter at the beginning of the procedure, so that changes can be detected at the end (recall the semantics of VAR parameters in section 2.1). If in addition the data type of the VAR parameter is an array or contains arrays, the translator must generate declarations for index variables of the appropriate types. These are used to traverse the data structure when the variable and its buffer undergo their componentwise comparison at the end (see below for details).

If the DOWN procedure had any CRITICAL procedure parameters, the master will send the name of the former CRITICAL procedure to the slave as described in section 3.7.2. What is actually sent is the static link and entry point for the corresponding procedure in the master. To be able to receive this information explicitly in the slave, the static link and entry point are treated as integers and the translator generates integer variable declarations for the purpose, and an RTS procedure redefinition to receive them.

The first statements that are executed in the slave procedure are the RTS calls to receive COPY variables. At each level of the nesting pattern in the slave procedure, the local COPY variables (if any) are received and the next level is activated. Finally, we reach the level of the DOWN procedure proper. At the beginning of the statement part at this level, the translator inserts the RTS calls to receive the DOWN procedure parameters. For VAR parameters, the RTS call is followed by an assignment statement to store a copy of the original data in the corresponding buffer variable. The user statements follow, unchanged except for explicit invocations of Pascal-C features, which on the slave side are limited to CRITICAL procedure calls.

In the Pascal-C source program, the DOWN procedure cannot contain any CRITICAL procedure declarations (not in a two-level system, at any rate), but it can call any of the CRITICAL procedures in its parameter list and have the corresponding actual procedures executed in the master. This remote invocation of a CRITICAL procedure is implemented by RTS calls, but it is nevertheless convenient for the translator to set up a local version of the CRITICAL procedure that will collect the parameters and issue the RTS calls. (Recall the analogous situation in section 3.7.2, where it was found convenient to have such local versions of DOWN procedures in the master module.) The modified procedure heading omits the keyword CRITICAL, and adds three value parameters to the original list, which is itself restricted by the language definition to value parameters (see section 2.3). The three new parameters give the total size of the original parameters and the static link and entry point, treated as integer variables, for the actual CRITICAL procedure residing in the master. This completes the heading for the local slave version of the CRITICAL procedure. Inside the block there is only one declaration and one statement. The declaration is an RTS redefinition to send the whole parameter list in a single block. ~~This~~ is basically for the convenience of the RTS, which will send the package up to the master as a CRITICAL procedure request. The sole statement is the call to the RTS procedure. The first parameter is the only one sent, but it is a VAR parameter and the size is the size of the whole parameter block, so the call effectively sends the whole parameter block to the RTS. The master RTS will eventually decompose the package and use it to simulate a local call to the former CRITICAL procedure.

At the end of the statement part of the DOWN procedure, the master inserts statements to detect and send back the updated portions of the VAR parameters. Each VAR parameter has a buffer variable containing a copy of its contents at the time of invocation; these two are compared component by component. ("Component" refers to simple data types or set types, which in this context are considered indivisible.) For each VAR parameter, there is an initial RTS call to announce the update and pass the starting address of the local variable. The subsequent calls give the size of each component in bytes and a boolean parameter to indicate whether or not that component has been altered. The actual boolean parameter is an expression of the form "variable_component <> buffer_component". To access all the components in an order that reflects their order of storage within the variable, the translator must insert statements to traverse the data structure in tree fashion, with the RTS calls occuring at the leaves. For records the fields are considered in turn in their usual order, while for arrays the translator inserts (possibly nested) loops using the variables set aside for indexing the arrays. The task of generating these statements is further complicated by the fact that the compiler used on the output modules will always allocate an even number of bytes to array and record structures, adding a dummy byte if necessary to make the number even. To keep the RTS's offset calculations accurate, the translator must check every structured component in the variable to determine if the component's size is odd, and if so it appends another RTS comparison call, a "dummy" call, with a value of "false" to indicate "no change" and a byte size of 1. In this way, the RTS can be spared the details of the variable's structure; the information in these calls is adequate for it to maintain a byte-by-byte rather than a component-by-component record of the updated portions. This allows the RTS to perform its complex updating tasks in the slaves and in the master in a fairly uniform fashion.

The final statement in the slave procedure is an RTS call to signal the completion of the DOWN procedure.

3.7.5. Synthesis strategy

It would be impractical to attempt the entire task of translation in a single pass. Consider, for example, that a procedure may be named in a COPY list long after its declaration has been processed. Clearly, to copy the procedure to the slave module the source text must be reread, at least up to the end of that procedure. An additional pass is therefore required to copy out the source segments that were identified in the first pass. (Technically, a second pass could be avoided by copying every procedure declaration into a separate file in case it later appears in a COPY section and must be merged into the output, but it is not difficult to see that, if anything, this alternative requires more overhead than a straightforward additional pass.) Of course the analysis conducted in the first pass need not be repeated, since the additional pass is required only for simple merging of the text.

While analyzing the program in the first pass, the translator generates a sequence of directives that will later be reread, in the same strictly sequential fashion, to direct the construction of the output modules. There are two kinds of directives to consider: one for copying from the source text and the other for inserting new text generated by the translator. The deletion of parts of the source text is performed implicitly, by issuing two copying directives that skip the portion in question between them. Insertion directives contain the actual text to be inserted; copying directives, in contrast, contain the coordinates in the source text for the beginning and end of the portion to be copied. Each coordinate consists of a line number and a character position within the line.

When the translator generates copying directives, it gets the coordinates from the current position, or from a saved position, or from a position recorded in the symbol table, depending on the circumstances. It is mainly for this reason that many objects in the symbol table have these coordinates as attributes.

The directives could be arranged so as to require only one additional pass. This would involve interleaving and combining the output so that no backing up is ever required while traversing the source module for the second time (many objects, such as procedures in a COPY list, end up in both master and slave modules). This in turn would complicate matters for the translator, since it would have to perform backpatching on the directives and keep open an arbitrary number of destination files. Instead, the current implementation issues a separate sequence of directives for each output module (not including the slave's main program, which is generated without the use of directives) and processes the sequences in turn, rereading the source program each time a new set of copying directives is opened. Within each sequence, of course, the directives are arranged so that only one rereading of the source program is necessary. Since only one module is created at a time, the destination module is known implicitly and need not appear as an argument in the directives.

Hence the total number of passes performed by this translator is N+2, where N is the number of DOWN procedure declarations in the source program, but only the first pass involves analysis of the text.

4. Software Tools for the Translator

This chapter describes several software tools that were used to help develop the Pascal-C translator, and should prove equally useful in maintaining it. Of these tools, only the parser generator was already available; the others were developed specially for this project.

The parser generator is a product called SLRGO developed at the University of Toronto. The input to SLRGO is a list of grammatical productions and its output is the corresponding set of tables for an SLR(1) parser [10,sec.6.3]. The parser driver, on the other hand, is hand-written and embedded within the translator as a procedure. It uses the data in the SLR(1) tables, together with the stream of input tokens, to determine the sequence of moves. Now the driver, like the rest of the translator, is implemented in Pascal, whereas the tables are produced as data declarations to be incorporated within an XPL program. One could still incorporate the tables into the translator by editing them or copying their contents, but this would be a tedious and error-prone operation. Moreover, the result would be to inflate the translator's code with either a large number of assignment statements in the initialization routine, or else a value clause that initializes static variables automatically, which has the additional disadvantage of being a nonstandard feature. A better solution is to choose whatever data structures are most appropriate for the kind of access required by the parser, and to convert the SLRGO tables to this form automatically. In this implementation, there is a separate program called PCCONV for this purpose so that the translator does not have to repeat the time-consuming process of conversion every time it is run. Instead, PCCONV produces a permanent file of binary data, PCTABL, that the translator reads in to initialize its data structures. Of course PCCONV must be executed (following SLRGO) whenever the grammar is modified.

In a syntax-directed translator such as this one, the parser controls semantic processing, lexical analysis, and syntactic error handling. But since the parsing tables are generated automatically, they contain rather arbitrary data that is difficult to link up meaningfully with those other functions. The semantic routines are selected according to the current reduce state, which is in turn related to the sequence number of the corresponding production. This leaves the linkage between reduce states and semantic actions highly vulnerable to changes in the grammar such as adding or deleting a production. Similarily, the parser generator uses its own code to represent grammatical tokens, and this must be linked to the representation used by the lexical analyzer which must deliver these tokens to the parser. Again, such a linkage is unstable because it is drastically affected by changes in the grammar. Finally, as explained in section 3.4, every syntactic error message in this translator is systematically derived from the kernel of the LR(0) set of items corresponding to the shift state at the time of detection. This requires not only a table to link shift states to kernel information, but also a table to link the parser generator's code for grammar symbols to their external string representations, so that the message can be constructed. As already noted, the symbol represenation used by the parser generator may change considerably with a slight change to the grammar.

For the sake of easier development and maintenance, the principle of automatic parser generation was extended to all features that ultimately depend on the grammar. The result is a multi-step procedure that must be followed after any change to the grammar, involving a number of programs developed for this purpose. It essentially factors out all syntactic aspects of the translator and unites them in this procedure. The end result is that the binary file that contains the converted parsing tables is expanded to include other initialization data, and that some text is generated that must be merged into the source text of the translator, overwriting the previous text at those points and requiring a recompilation of the translator.

The remainder of this chapter outlines the steps involved in this procedure.

Step 1 - Linking syntax and semantics

The starting file contains the productions of the grammar in the form required by the SLRGO parser generator. It also contains the semantic statements (in Pascal), most of which are calls to semantic procedures. Each group of statements appears just after the production to which it should be linked. Some productions, of course, may not be associated with any semantic actions.

This file is built and maintained manually. A program called PCGRAM reads it and produces two output files. The first consists of the productions alone; it is used as the input to SLRGO in the next step. The other contains the core of a Pascal CASE statement. This statement is used to select the appropriate semantic actions based on the CASE selector, which is the production number indicated by the current reduce state. Each CASE element contains the CASE constant, which represents the production number, followed by the text of the production as a comment for the sake of legibility, and finally the associated semantic statements. This text is merged into the source code of the translator inside the semantic-control routine.

Step 2 - Generating the parser

The file containing the grammatical productions is used as input to SLRGO. If the grammar is not SLR(1) the generator may still work but the output must be examined to determine whether the default parsing actions are correct. If not, either the tables must be altered manually, or else the process must be repeated from step 1 with a modified grammar. At present the grammar has several non-SLR(1) features but all of them can be safely ignored except for one that requires manual intervention. This is the familiar ambiguity regarding the two forms of the IF statement: with and without the ELSE clause. Removing this ambiguity at its source requires substantial complications in the grammar. In contrast, its effect can be corrected by simply removing ELSE from the look-ahead set of the IF statement in the tables. This is fairly simple to do manually if care is taken.

SLRGO recognizes a number of user options. One of these produces a listing that includes the LR(0) sets of items, which are needed to compose syntactical error messages. Unfortunately, the listing of the parser generator is the only source of the LR(0) items, so the information must be extracted from this textual form. Both the XPL parsing tables and the listing serve as input to step 3.

Step 3 - Producing the final tables

The primary task for program PCCONV is to convert the XPL tables produced in step 2 and output the results in binary form to the initialization file PCTABL. PCCONV then reads the listing produced in step 2 by SLRGO, and thereby obtains the kernel items. The table relating kernel items to shift states is added to the initialization file for use in generating sytactic error messages. Another table added to the initialization file links the external representations of the grammar symbols to their internal values as assigned by the parser generator. This completes the translator's capability for generating syntactic error messages.

As it produces the binary tables contained in PCTABL, PCCONV also generates the text for certain constant definitions. These are used to define the limits of the data structures in PCTABL. They will be merged into the translator's source text in step 4.

The program then opens another input file called PCSYMB. Like PCGRAM in step 1, PCSYMB is a manually created text file. This file features two lists of items in a relatively free format. The first is a list of all terminal symbols that are referenced by name in the translator. A constant definition is produced for each one and added to the previous file of constant definitions. The external representation appears on the left-hand side of the definition and the internal integer value assigned by the parser generator appears on the right. Hence the symbol can be referenced anywhere by name and yet be linked to its representation in the parsing tables. In cases where the external representation does not qualify as a Pascal identifier it is even possible to specify a pseudonym that will appear in its place on the left-hand side with the same internal value on the right. (Only strings appear in this input file; the internal values used by the parser generator are determined automatically by PCCONV by matching the strings.)

The second part of file PCSYMB is a list of all terminal symbols that are keywords in the language. The translator does not refer to these directly, since in this implementation keywords are stored like ordinary identifiers in the symbol table, but the lexical analyzer must be able to return the correct internal value when it recognizes a keyword. The external and internal representations of the keywords are added to PCTABL, which is now complete. During initialization, the compiler reads the keywords from PCTABL and stores them in the symbol table along with their internal values.

Step 4 - Editing and compiling

Finally, the constant definitions produced in step 3 and the case statement list produced in step 1 are merged, with the help of a text editor, into the appropriate places in the source program, taking care to overwrite the superseded text. The translator is then given over to the Pascal compiler.

5. Scheduling Processes at Run Time

As explained in chapter 3, the Pascal-C translator produces a set of modules in sequential Pascal [4]. These modules do not themselves display any concurrency, but rather invoke the Pascal-C run-time system [5] through external procedure calls. It is the run-time system (and its supporting software, such as the communication subsystem) that implements Pascal-C's concurrent features. Most of the RTS for Pascal-C is written in Parallel Pascal [13], whose concurrent features are inspired by Modula [12].

There are some aspects of Parallel Pascal's run-time support that are not well suited for implementing Pascal-C, notably the scheduling policy. Since Parallel Pascal's RTS was distributed as a package of source modules (in Macro-11 assembler language [14]), it was possible to make the required modifications. This chapter outlines the reasons for and extent of these changes to Parallel Pascal's RTS. The full details can be found in [15]. Note that most of the comments regarding Parallel Pascal also apply to its predecessor Modula.

We will begin by reviewing the varieties of concurrency found in the Pascal-C system. In the first place, we have the master program and its DOWN procedures. These processes run on entirely separate hardware units and therefore interact in a rather indirect manner. Each unit treats the others connected to it as peripheral devices that may initiate action at essentially unpredictable times. This is commonly encountered in operating systems with I/O devices and suggests a similar solution. Namely, for each peripheral device there is a "device handler", a concurrent process dedicated to servicing it [16,p.69]. This is an apparent rather than a true concurrency. Such processes execute in interleaved fashion on the same processor, unlike the truly parallel execution among the DOWN procedures and the master program. It is also worth noting that the communication processes are permanent, not created dynamically like DOWN-procedure activations. They belong to the run-time system, their activities and data structures transparent to the Pascal-C programmer. For example, in the master we would have one or more processes to receive communications from the slaves. The actual parameter updates returned at the end of a DOWN procedure would be stored in a buffer; a CRITICAL procedure request would be added to a queue of such requests. Being transparent to the user program, such actions can be performed asynchronously.

It is in fact important that these tasks be completed as soon as possible, so that the slaves can go on with other useful work. We again note an analogy with operating systems, where I/O devices are a bottleneck and must be kept busy to exploit their potential parallelism, to which end priority is often given to jobs that are I/O bound. In the Pascal-C system, priority should be given to the communication processes to keep the slaves as busy as possible. The user program should be suspended in favour of the communication processes whenever possible.

A CRITICAL procedure requested by a slave is another process, distinct from both the master process and the communication processes just mentioned. Since the semantics of Pascal-C demand the serialization of CRITICAL procedures, there need be only one process in the system dedicated to running them. This process may be viewed as the consumer in a producer-consumer relationship with the process that queues CRITICAL procedure requests. It would fit in best with the semantics of Pascal-C if it were given a priority intermediate between the low-priority master process and the high-priority communication processes. One peculiarity of this process is that, unlike the communication processes, it has access to practically the entire memory space of the master process. (In the present implementation, this process does not run the CRITICAL procedure directly. Instead, it calls a special routine that manipulates the description and stack of the main process so that when the main process resumes, it will simulate an ordinary call to the CRITICAL procedure.)

So the scheduling policy should support a priority ordering of processes. Unfortunately, Parallel Pascal does not support priority-based scheduling. As each process is created, it is linked into a circular queue, and the processes retain their relative ordering throughout their lifetimes. When the currently active process waits for a signal, control passes to the next runnable process in the queue. If it sends a signal, on the other hand, and some waiting process is thereby unblocked, control passes to that process. The authors of Parallel Pascal call this round-robin scheduling, but it is clear from the above description that this is only true in the case of the wait statement. Consider what happens if we have three processes A, B, and C in the circular queue, with A periodically waiting for a signal that is later sent by B. If B immediately follows A in the queue, control will slide to B when A waits, and will pass back to A when B sends, jumping over C. As long as this pattern persists, C will starve. But given a different order of creation, C might be the next process in the queue after A, in which case starvation would be avoided. This scheduling policy is evidently unsatisfactory in its own right, quite apart from the lack of priority capabilities.

In the revised run-time system for Parallel Pascal, a process sets (or alters) its own priority by calling a special procedure. The circular queue which formerly contained all normal processes is replaced by a linear one which contains only those that are runnable. The processes in this "ready" queue are linked in descending order of priority. At the top is the next one to run if a switch takes place, while at the bottom is a special idle task that always has the lowest priority, in case all normal processes are blocked. The currently active process is not in the queue. When a process is inserted into the queue, it is ordinarily placed behind those of greater or equal priority. This amounts to a round-robin scheduling policy among processes of equal priority. In this case, a process's position among its peers in the queue depends upon how long it has been runnable but idle. This is "fairer" than the round-robin ordering in the original version, which depended solely on the relative order of the processes' creation. Whenever the current process sends a signal, it is suspended and re-inserted into the ready queue to allow other processes of the same priority to run. The signalled process (if any) is not automatically activated as it formerly was, since it too is inserted behind other ready processes of its priority. The ready queue thus resembles a signal queue in Parallel Pascal. (A process that issues a wait on a signal is inserted into the signal queue behind all other processes of the same "rank", which is specified as a parameter to Parallel Pascal's wait procedure.)

So a process that sends a signal will be suspended as before, but the process that takes its place will be chosen according to consistent scheduling criteria. In retrospect, it may be preferable to drop altogether the practice of automatically suspending a process when it sends a signal. Sending a signal and self-suspension are distinct operations that may not be both appropriate in a given situation. This is implicitly recognized in Parallel Pascal in the case of interrupt routines, which are never suspended when they send a signal. It would be convenient to make this kind of send available to normal processes as well, with perhaps a separate call to provoke a round-robin shift, with no reference to any signal, whenever appropriate.

Since an interrupt routine can send a signal, it may unblock a process of higher priority than the one that was running when the interrupt occured. Yet Parallel Pascal always restores the interrupted process, regardless of the situation when the interrupt routine finishes. This practically defeats the usefulness of a priority scheme, since the high-priority process will remain idle indefinitely - that is, until the current process suspends itself, through a wait or send for example. Accordingly, the exit from an interrupt routine had to be changed to permit a scheduling action when appropriate. This task was somewhat complicated by the fact that interrupt routines in PDP-11 systems [17] may themselves be interrupted, and no normal process should be restored until the lowest-nested interrupt has been serviced. User-written interrupt routines can employ a counter to keep track of the level of nesting, but some interrupt routines are hidden within the RT-11 operating system [17]. It appeared that the only solution was to let the RT-11 system keep track of the level of nesting, which is done by following certain conventions, mainly a set of system calls at the beginning and end of all interrupt routines. When a switch is appropriate, the interrupt routine makes a special RT-11 call that schedules a section of user code for execution following the lowest-level interrupt routine. It is this serialized code that effects the desired switch.

A number of improvements which are not related directly to schedulng were made as the opportunities arose. The most important ones are described below. A variety of other deficiencies in the original version were corrected or at least documented in [15].

Wait and send are supposed to be performed differently when issued by an interrupt routine. The distributed version of Parallel Pascal relies on the compiler alone to make this distinction: All waits and sends are translated into their normal forms unless they appear within the statement part of an interrupt routine. It is therefore an error for an interrupt routine to call a procedure that calls wait or send, but this error is not detected and may bring down the system eventually. In the revised version, the scheduler precedes any action by a simple run-time test that determines whether a normal or interrupt process is issuing the statement.

Certain operations in the run-time system manipulate important global data; procedures new and dispose, for example, alter heap markers. In the distributed version of Parallel Pascal, these operations are vulnerable to re-entry via interrupt routines. In the revised version, they are considered critical regions and can only be performed by one process at a time. Inside the critical regions, no process switching is allowed. Interrupts may occur, but any high-priority process that may be unblocked by the interrupt routine is only activated when the current process exits the critical region. If the interrupt routine itself attempts to enter a critical region (which would not be very appropriate since most of the operations concerned are rather too lengthy to be performed as part of an interrupt service), the routine is suspended and forced to resume as a normal process for the remainder of the interrupt servicing, subject to the usual scheduling rules. These measures ensure the serialization of all processes with respect to critical regions.

I/O operations are also treated as critical regions. Formerly, most I/O operations performed an implicit wait, allowing other processes to run before the I/O operation was complete. In the revised version the process does not lose control when it performs I/O.

When an interrupt routine issues a wait, it is suspended and treated as a normal process for the remainder of the interrupt servicing, just as in the case of an interrupt routine that attempts to enter a critical region. This approach in the revised version makes for more uniform process handling in the scheduler, and removes the former restriction against one interrupt routine signalling another (which formerly resulted in an undiagnosed system failure).

The signal mechanisms provided by Parallel Pascal can lead to the loss of a signal if a signal is sent but no process is waiting for it. The revised version uses the signal mechanisms as a basis for developing Dijkstra-type semaphores [18]. A semaphore has a count associated with it. Each send increments the count if no process is waiting for it at the time. The next process to issue a wait on the semaphore will decrement the count instead of actually waiting. These operations are protected against any interrupts and may therefore be considered primitives.

A few low-level routines specifically intended for Pascal-C had to be implemented in addition to the general-purpose ones described above. One of them simulates an ordinary call to a CRITICAL procedure from the master process. Another one saves crucial information at a stable point in a slave's main program so that its status may be restored in the case of a TERMINATE command from the master (see section 3.7.3). A complementary routine, invoked by the slave RTS to complete termination, restores the status recorded by the first routine and therby returns the slave to the idle state, ready for another DOWN procedure activation.

6. Conclusion

At the present stage of the Pascal-C project, we have a working prototype that has demonstrated the feasibility of solving combinatorial problems on an inexpensive multiprocessor, using a powerful applications-oriented language based on Pascal.

The main part of this thesis described the translator for the Pascal-C language. In the prototype system, this translator generates sequential Pascal, but its structure is sufficiently modular to permit the eventual addition of low-level code generation without unduly affecting the other components of the translator. One notable aspect of the translator's design is the attempt to include systematic strategies for reporting and recovering from syntactic errors, problems which persist in compiler design despite the development of systematic methods for parsing valid programs. The performance of these error strategies has so far fulfilled expectations. Certain software tools were also developed that effectively extend the scope of the automatic parser generator, linking together all components of the translator that are associated with the syntax, including the error-handling mechanisms just mentioned. These tools ease the chore of developing and maintaining the translator. The revised scheduling system described in chapter 5 also seems to ~~be~~ performing adequately.

On the negative side, the lack of low-level code generation in the prototype system is bound to prove an inconvenience to users, since this adds another step to the already tedious process of preparing a source program for execution on the multiprocessor. Moreover, the translator's output modules, along with most of the RTS routines, are compiled using Parallel Pascal, whose compile-time and run-time error diagnostics are woefully inadequate. These shortcomings should disappear if and when a complete compiler and RTS are developed for Pascal-C.

With regard to the language features, it was the semantics of VAR parameters to DOWN procedures that presented the most difficulties. The translator has to construct elaborate code in the slave procedures to determine which components of the parameters have been altered, and must insert code in many of the master's procedures to check against the premature deallocation of the actual VAR parameters. Similar challenges would present themselves in any implementation of Pascal-C, though it might be possible to shift some of the burden to the RTS.

As noted in the text, some of the original language specifications were changed, partly in consideration of the relative ease of implementation. The variables in the COPY section, for example, must have been declared with single type identifiers, but this is of importance only in the prototype version. Other changes have merits of their own, and should perhaps be considered as permanent language features. The inclusion of types and constants in the COPY section has the appeal of consistency, while the provision of CRITICAL procedure parameters to DOWN procedures provides more flexibility and avoids forward referencing.

REFERENCES

1. J.E. Hopcroft and J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

2. C. Lam, J.W. Atwood, S. Cabilio, B.C. Desai, P. Grogono, J. Opatrny, "A multiprocessor project for combinatorial computing", CIPS Session 82, 1982.

3. B.C. Desai, C. Lam, J.W. Atwood, J. Opatrny, P. Grogono, S. Cabilio, "NOVAC - A non-tree variable tree for combinatorial computing", Proc. of 11th Int. Conf. on Par. Proc., 1982.

4. D. Cooper, Standard Pascal User Reference Manual, W.W. Norton, 1983.

5. Y. Wong, "The implementation of the run-time system of a concurrent programming language, Pascal-C", M.Comp.Sci. Thesis, Dept. of Comp. Sci., Concordia U., 1985.

6. P. Brinch-Hansen, "The programming language Concurrent Pascal", IEEE-TSE 1, 2, 1975, pp. 199-207.

7. J.W. Atwood, S. Ganesan, M. Lafleur, W. Prager, "Measurements of the solution of two combinatorial problems", CIPS Session 84, 1984.

8. P. Grogono, Pascal-C Report, Concordia U., 1983.

9. F.L. DeRemer, "Lexical analysis", from Compiler Construction: An Advanced Course, F.L. Bauer and J. Eickel, ed., Springer-Verlag, 1974.

10. A.V. Aho and J.D. Ullman, Principles of Compiler Design, Addison-Wesley, 197..

11. A.V. Aho and S.C. Johnson, "LR parsing", Computing Surveys, 6, 2, 1974.

12. N. Wirth, "Modula: a language for modular multiprogramming", Software Practice and Experience, 7, 1, 1977.

13. Parallel Pascal User's Manual, Interactive Technology Inc., Portland, Oregon, 1982.

14. PDP-11 Macro-11 Language Reference Manual, Digital Equipment Corporation, Maynard, Mass., 1983.

15. S. Cabilio, Parallel Pascal - Version 2, Concordia U., 1984.

16. A.M. Lister, Fundamentals of Operating Systems, (3rd ed.), Macmillan, 1984.

17. RT-11 Software Support Manual, Digital Equipment Corporation, Maynard, Mass., 1983.

18. E.W. Dijkstra, "Cooperating sequential processes", from Programming Languages (ed. F. Genuys), Academic Press, 1968.

APPENDIX 1.1

```
(*   This is a solution to the knapsack problem in Pascal.
S [FIRST..LAST] contains a list of integers. The program
finds out if any subset of them adds up to the integer T
and either outputs the first such subset found or else
reports failure to find a solution.

     The heart of the program is the recursive procedure TRY
which attempts every subset CHOICE of the integers in S
until a solution is found or else all possibilities have
been exhausted.

     Note the GOTO in procedure SOLUTION, which has the
effect of jumping out of the recursive sequence when a
solution is found. This makes the sequential version of
SOLUTION somewhat analogous to the CRITICAL version in
appendix 1.2. *)


PROGRAM KNAPSACK1 (INPUT, OUTPUT);

   LABEL
      99;

   CONST
      FIRST = 0;
      MAX   = 50;

   TYPE
      ITEMTYPE = FIRST .. MAX;
      SUBSET   = SET OF ITEMTYPE;
      SLIST    = ARRAY [FIRST..MAX] OF INTEGER;

   VAR
      LAST : ITEMTYPE;
      ITEM : ITEMTYPE;
      T    : INTEGER;
      S    : SLIST;
      FOUND: BOOLEAN;
```

```
PROCEDURE SOLUTION (CHOICE : SUBSET);

   VAR
     ITEM : ITEMTYPE;

   BEGIN (* SOLUTION *)
      WRITELN ('HERE IS A SOLUTION:');
      FOR ITEM := FIRST TO LAST DO
         IF ITEM IN CHOICE THEN
            WRITELN ('ITEM: ',ITEM,'  VALUE: ',S [ITEM]);
      FOUND := TRUE;
      GOTO 99
   END;  (* SOLUTION *)

PROCEDURE TRYSUM (SUM : INTEGER; CHOICE : SUBSET;
                  ITEM : ITEMTYPE);

   BEGIN (* TRYSUM *)
      IF SUM = T THEN
         SOLUTION (CHOICE)
      ELSE
         IF ITEM <= LAST THEN
            BEGIN
               TRYSUM (SUM, CHOICE, SUCC (ITEM));
               TRYSUM (SUM +S [ITEM], CHOICE +[ITEM],
                       SUCC (ITEM))
            END
   END;  (* TRYSUM *)

BEGIN (* KNAPSACK1 *)
   READ (LAST, T);
   FOR ITEM := FIRST TO LAST DO
      READ (S [ITEM]);
   FOUND := FALSE;
   TRYSUM (0, [], FIRST);
99:
   IF NOT FOUND THEN
      WRITELN ('THERE IS NO SOLUTION')
END.  (* KNAPSACK1 *)
```

APPENDIX 1.2

```
(*   This is a Pascal-C version of the knapsack
program of appendix 1.1. It illustrates a typical use of
CRITICAL procedures to return results and to TERMINATE the
DOWN procedures in the slaves.

     At a given level (variable DEPTH) in the
recursion, the subproblems are consigned to slaves
by calling the DOWN procedure SLAVETRY. Note several
identifiers that had to be included in SLAVETRY's COPY
section.

     There are now two recursive procedures in the program,
one for the master and the other for the slaves.

     MASTERTRY performs the upper levels of the recursion
in the master, calling SLAVETRY at the proper depth.
TRY is identical to its namesake in appendix 1.1, and
performs the lower levels of recursion in the slaves.

     SOLUTION is now a CRITICAL procedure called by TRY
from the slaves. It prints the solution as before, and
terminates all activations of SLAVETRY. *)


PROGRAM KNAPSACK2 (INPUT, OUTPUT);

   CONST
      FIRST = 0;
      MAX   = 50;

   TYPE
      ITEMTYPE = FIRST .. MAX;
      SUBSET   = SET OF ITEMTYPE;
      SLIST    = ARRAY [FIRST..MAX] OF INTEGER;

   VAR
      LAST : ITEMTYPE;
      ITEM : ITEMTYPE;
      T    : INTEGER;
      S    : SLIST;
      FOUND: BOOLEAN;
      DEPTH: INTEGER;
```

```
DOWN PROCEDURE SLAVETRY (SUM : INTEGER; CHOICE : SUBSET;
          CRITICAL PROCEDURE SOLUTION (CHOICE : SUBSET));

   COPY
      SUBSET, LAST, T, S, DEPTH;

   PROCEDURE TRYSUM (SUM : INTEGER; CHOICE : SUBSET;
                     ITEM : ITEMTYPE);

      BEGIN (* TRYSUM *)
         IF SUM = T THEN
            SOLUTION (CHOICE)
         ELSE
            IF ITEM <= LAST THEN
               BEGIN
                  TRYSUM (SUM, CHOICE, SUCC (ITEM));
                  TRYSUM (SUM +S [ITEM], CHOICE +[ITEM],
                          SUCC (ITEM))
               END
      END;  (* TRYSUM *)

   BEGIN (* SLAVETRY *)
      TRYSUM (SUCC (DEPTH), SUM, CHOICE)
   END;  (* SLAVETRY *)

CRITICAL PROCEDURE SOLUTION (CHOICE : SUBSET);

   VAR
      ITEM : ITEMTYPE;

   BEGIN (* SOLUTION *)
      WRITELN ('HERE IS A SOLUTION:');
      FOR ITEM := FIRST TO LAST DO
         IF ITEM IN CHOICE THEN
            WRITELN ('ITEM: ',ITEM,'  VALUE: ',S [ITEM]);
      FOUND := TRUE;
      TERMINATE (SLAVETRY)
   END;  (* SOLUTION *)
```

```
PROCEDURE MASTERTRY (SUM : INTEGER; CHOICE : SUBSET;
                     ITEM : ITEMTYPE);

   BEGIN (* MASTERTRY *)
      IF ITEM > DEPTH THEN
         SLAVETRY (SUM, CHOICE, SOLUTION)
      ELSE
         BEGIN
            MASTERTRY (SUM, CHOICE, SUCC (ITEM));
            MASTERTRY (SUM +S [ITEM], CHOICE +[ITEM],
                       SUCC (ITEM))
         END
   END;  (* MASTERTRY *)


BEGIN (* KNAPSACK2 *)
   READ (LAST, T, DEPTH);
   FOR ITEM := FIRST TO LAST DO
      READ (S [ITEM]);
   FOUND := FALSE;
   MASTERTRY (0, [], FIRST);
   WAIT (SLAVETRY);
   IF NOT FOUND THEN
      WRITELN ('THERE IS NO SOLUTION')
END.  (* KNAPSACK2 *)
```

APPENDIX 1.3

```
(*  This is the master module produced-by the translator
for the Pascal-C version of knapsack in appendix 1.2.
Note that this module, like the ones in appendices
1.4, 1.5, 2.3, 2.4, and 2.5, were actually produced
by the translator but have been edited somewhat to
fit them into the present page format. *)


PROGRAM KNAPSACK2 (INPUT, OUTPUT);
VAR ZZRTSBUF:ARRAY [1..1500] OF INTEGER;(*FOR PASCAL-C RTS*)


   CONST
      FIRST = 0;
      MAX   = 50;

   TYPE
      ITEMTYPE = FIRST .. MAX;
      SUBSET   = SET OF ITEMTYPE;
      SLIST    = ARRAY [FIRST..MAX] OF INTEGER;

   VAR
      LAST  : ITEMTYPE;
      ITEM  : ITEMTYPE;
      T     : INTEGER;
      S     : SLIST;
      FOUND: BOOLEAN;
      DEPTH: INTEGER;

   (*PASCAL-C RTS DEFINITIONS*)
PROCEDURE RTS1 (ACTION:INTEGER); EXTERNAL;
PROCEDURE RTS2 (ACTION,DATA:INTEGER); EXTERNAL;
PROCEDURE RTS3 (ACTION,DATA1,DATA2:INTEGER); EXTERNAL;
FUNCTION RTS4 (ACTION,DATA:INTEGER):BOOLEAN; EXTERNAL;
(*END RTS DEFINITIONS*)

 PROCEDURE SLAVETRY (SUM : INTEGER; CHOICE : SUBSET;
             PROCEDURE SOLUTION (CHOICE : SUBSET));
(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:ITEMTYPE;
                ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:ITEMTYPE;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
```

```
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO3(ZZACTION:INTEGER;VAR ZZDATA:SLIST;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:SLIST;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO4(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO5(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO6(ZZACTION:INTEGER;VAR ZZDATA:ITEMTYPE;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:ITEMTYPE;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO7(ZZACTION:INTEGER;
                 PROCEDURE ZZCRIT(CHOICE:ITEMTYPE));
PROCEDURE RTS5(ZZACTION:INTEGER;
               PROCEDURE ZZCRIT(CHOICE:ITEMTYPE));EXTERNAL;
BEGIN RTS5(ZZACTION,ZZCRIT)END;
(*END RTS REDEFINITIONS*)

BEGIN
IF RTS4(96,1)THEN BEGIN RTS3(66,1,0);(*NEW ACTIVATION UNLESS
                                       TERMINATED*)
ZZPRO1(64,LAST,1); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO2(64,T,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO3(64,S,102); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO4(64,DEPTH,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO5(64,SUM,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO6(64,CHOICE,1); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO7(128,SOLUTION); (*SEND CRIT PROC INFO TO SLAVE*)
RTS1(16)END; (*END DOWNLOADING*)
END;  (* SLAVETRY *)

    PROCEDURE SOLUTION (CHOICE : SUBSET);

      VAR
         ITEM : ITEMTYPE;

      BEGIN (* SOLUTION *)
         WRITELN ('HERE IS A SOLUTION:');
         FOR ITEM := FIRST TO LAST DO
            IF ITEM IN CHOICE THEN
```

```
          WRITELN ('ITEM: ',ITEM,'  VALUE: ',S [ITEM]);
      FOUND := TRUE;
      RTS2(33,1) (*TERMINATE (SLAVETRY)*)

   END;  (* SOLUTION *)

  PROCEDURE MASTERTRY (SUM : INTEGER; CHOICE : SUBSET;
                       ITEM : ITEMTYPE);


PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);
PROCEDURE RTS2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);
               EXTERNAL;
BEGIN RTS2(ZZACTION,ZZDATA)END;
BEGIN (* MASTERTRY *)
         IF ITEM > DEPTH THEN
            BEGIN RTS2(34,1);RTS1(0); (*RESERVE SLAVE & LOCK
                                         BEFORE DOWN CALL*)
SLAVETRY (SUM, CHOICE, SOLUTION);RTS1(1)END (*UNLOCK AT END
                                               OF DOWN CALL*)

         ELSE
            BEGIN
               MASTERTRY (SUM, CHOICE, SUCC (ITEM));
               MASTERTRY (SUM +S [ITEM], CHOICE +[ITEM],
                          SUCC (ITEM))
            END
      ;ZZPRO1(35,SUM); (*CHECK FOR ACTUAL VAR PARAMS TO DOWN
                          PROCS*)
END;  (* MASTERTRY *)


  BEGIN (*MASTER MAINLINE*)
RTS2(48,1); (*INITIALIZE MASTER RTS*)
 (* KNAPSACK2 *)
      READ (LAST, T, DEPTH);
      FOR ITEM := FIRST TO LAST DO
         READ (S [ITEM]);
      FOUND := FALSE;
      MASTERTRY (0, [], FIRST);
      RTS2(32,1) (*WAIT (SLAVETRY)*)
;
      IF NOT FOUND THEN
         WRITELN ('THERE IS NO SOLUTION')
   ;RTS1(17) (* END PROGRAM *)
END.
```

APPENDIX 1.4

(*   This is the main module for the slave, created by
the translator for the Pascal-C version of knapsack. *)

```
VAR RTS:ARRAY [1..1500] OF INTEGER;(*FOR PASCAL-C RTS*)
    DOWNID:INTEGER;
PROCEDURE DP1;EXTERNAL;
PROCEDURE ZZSTAT;EXTERNAL;
PROCEDURE RTS2(X,Y:INTEGER);EXTERNAL;
PROCEDURE RTS6(X:INTEGER;VAR Y:INTEGER);EXTERNAL;
BEGIN
 RTS2(48,0); (* INITIALIZE SLAVE RTS *)
 ZZSTAT; (* SAVE STATE TO RESTORE AFTER TERMINATION *)
 REPEAT
  RTS6(160,DOWNID); (* WAIT FOR DOWN PROC REQUEST *)
  CASE DOWNID OF
   1:DP1;
   OTHERWISE
  END;
 UNTIL DOWNID = -1
END.
```

APPENDIX 1.5

(*   This is the external module for DOWN procedure
SLAVETRY in the Pascal-C version of knapsack. *)

```
(*$E*)PROCEDURE DP1;

CONST
FIRST = 0;
CONST
MAX   = 50;
TYPE
ITEMTYPE = FIRST .. MAX;
TYPE
SUBSET   = SET OF ITEMTYPE;
TYPE
SLIST    = ARRAY [FIRST..MAX] OF INTEGER;
VAR LAST:ITEMTYPE;
VAR T:INTEGER;
VAR S:SLIST;
VAR DEPTH:INTEGER;
(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:ITEMTYPE;
                  ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:ITEMTYPE;
                ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                  ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO3(ZZACTION:INTEGER;VAR ZZDATA:SLIST;
                  ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:SLIST;
                ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO4(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                  ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
(*END RTS REDEFINITIONS*)


PROCEDURE DOWN;
(*BEGIN DECLARATIONS FOR DOWN PROCEDURE PARAMETERS*)
VAR SUM:INTEGER;(*VALUE PARAMETER*)
VAR CHOICE:ITEMTYPE;(*VALUE PARAMETER*)
VAR ZZLIN7,ZZENT7:INTEGER; (*STAT LINK & ENTRY PT FOR
                             CRIT PROC IN MASTER*)
(*END OF DECLARATIONS FOR DOWN PROCEDURE PARAMETERS*)


      PROCEDURE TRYSUM (SUM : INTEGER; CHOICE : SUBSET;
                        ITEM : ITEMTYPE);

         BEGIN (* TRYSUM *)
            IF SUM = T THEN
               SOLUTION (CHOICE,2,ZZLIN7,ZZENT7)

            ELSE
               IF ITEM <= LAST THEN
                  BEGIN
                     TRYSUM (SUM, CHOICE, SUCC (ITEM));
                     TRYSUM (SUM +S [ITEM], CHOICE +[ITEM],
                             SUCC (ITEM))
                  END
         END;  (* TRYSUM *)


(*PASCAL-C RTS DEFINITIONS*)
PROCEDURE RTS1(ACTION:INTEGER);EXTERNAL;
PROCEDURE RTS3(ACTION:INTEGER;CHANGED:BOOLEAN;
               SIZE:INTEGER);EXTERNAL;
(*END RTS DEFINITIONS*)

(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO5(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO6(ZZACTION:INTEGER;VAR ZZDATA:ITEMTYPE;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:ITEMTYPE;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO7(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE SOLUTION([illegible]CE:ITEMTYPE;
                   [illegible]IZE,ZZLINK,ZZENTRY:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(76,ZZENTRY,ZZSIZE+6)END;
```

```
(*END RTS REDEFINITIONS*)

BEGIN (*DOWN PROCEDURE CORE*)
ZZPRO5(74,SUM,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO6(74,CHOICE,1);(*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO7(74,ZZLIN7,4); (*RECEIVE CRIT PROC INFO FROM MASTER*)
 (* SLAVETRY *)
         TRYSUM (SUCC (DEPTH), SUM, CHOICE)
      RTS1(17)END; (* END DOWN PROCEDURE CORE *)

BEGIN (*REGION FOR COPY SECTION ITEMS*)
ZZPRO1(74,LAST,1); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO2(74,T,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO3(74,S,102); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO4(74,DEPTH,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
DOWN END; (*END COPY SECTION REGION*)
```

APPENDIX 2.1

```
(*  This is a Pascal version of quicksort.
The elements to be sorted are held in E [1..NUM].

     The heart of the program is the recursive procedure
SORT, which divides a given portion of E into two parts
that are sorted with respect to each other, then sorts each
of the two parts internally. SORT calls procedure PARTITION
to divide the portion into two and group lower values in
one part and higher values in the other. *)


PROGRAM QUICKSORT1 (INPUT, OUTPUT);

   TYPE
      ELEMENT = INTEGER;
      LIST    = ARRAY [1 .. 1000] OF ELEMENT;

   VAR
      E       : LIST;
      NUM, J  : INTEGER;

   PROCEDURE PARTITION (VAR E : LIST; LO, HI : INTEGER;
                        VAR MID : INTEGER);

      VAR
         KEY, TEMP : ELEMENT;
         K : INTEGER;
   BEGIN (* PARTITION *)
      KEY := E [LO];
      K   := LO;
      REPEAT
         REPEAT
            LO := LO+1;
         UNTIL E [LO] >= KEY;
         REPEAT
            HI := HI-1;
         UNTIL E [HI] <= KEY;
         IF LO < HI THEN
            BEGIN
               TEMP   := E [LO];
               E [LO] := E [HI];
               E [HI] := TEMP
            END
      UNTIL LO >= HI;
      E [K]  := E [HI];
      E [HI] := KEY;
      MID    := HI
   END;  (* PARTITION *)

PROCEDURE SORT (VAR E : LIST; LO, HI : INTEGER);

   VAR
      MID : INTEGER;

   BEGIN (* SORT *)
      IF LO < HI THEN
         BEGIN
            PARTITION (E, LO, HI+1, MID);
            SORT (E, LO, MID-1);
            SORT (E, MID+1, HI)
         END
   END;  (* SORT *)

BEGIN (* QUICKSORT1 *)
   READ (NUM);
   FOR J := 1 TO NUM DO
      READ (E [J]);
   E [NUM+1] := MAXINT;
   SORT (E, 1, NUM);
   FOR J := 1 TO NUM DO
      WRITE (E [J])
END.  (* QUICKSORT1 *)
```

APPENDIX 2.2

(* This is a Pascal-C version of the quicksort program of appendix 2.1. It illustrates the use of VAR parameters to return results from DOWN procedures in the slaves.

When the master creates portions of the list that are neither too small nor too large, it sends them to the slaves using DOWN procedure SLAVESORT. SLAVESORT, using the VAR mechanism, returns only those portions that have been sorted to update the original list in the master.

There are again two recursive procedures: MASTERSORT to handle the largest and smallest pieces in the master, and SORT to handle the protions sent to the slaves.

Note how the COPY section is used to avoid duplication of code by making procedure PARTITION available to the DOWN procedure. *)

```
PROGRAM QUICKSORT2 (INPUT, OUTPUT);

   TYPE
      ELEMENT = INTEGER;
      LIST   = ARRAY [1 .. 1000] OF ELEMENT;

   VAR
      E : LIST;
      NUM, J : INTEGER;
      LARGE, SMALL : INTEGER;

   PROCEDURE PARTITION (VAR E : LIST; LO, HI : INTEGER;
                        VAR MID : INTEGER);

      VAR
         KEY, TEMP : ELEMENT;
         K : INTEGER;
```

```
BEGIN (* PARTITION *)
   KEY := E [LO];
   K   := LO;
   REPEAT
      REPEAT
         LO := LO+1;
      UNTIL E [LO] >= KEY;
      REPEAT
         HI := HI-1;
      UNTIL E [HI] <= KEY;
      IF LO < HI THEN
         BEGIN
            TEMP    := E [LO];
            E [LO] := E [HI];
            E [HI] := TEMP
         END
   UNTIL LO >= HI;
   E [K]  := E [HI];
   E [HI] := KEY;
   MID    := HI
END;  (* PARTITION *)

DOWN PROCEDURE SLAVESORT (VAR E : LIST; LO, HI : INTEGER);

   COPY
      PARTITION;

   PROCEDURE SORT (VAR E : LIST; LO, HI : INTEGER);

      VAR
         MID : INTEGER;

      BEGIN (* SORT *)
         IF LO < HI THEN
            BEGIN
               PARTITION (E, LO, HI+1, MID);
               SORT (E, LO, MID-1);
               SORT (E, MID+1, HI)
            END
      END;  (* SORT *)

   BEGIN (* SLAVESORT *)
      SORT (E, LO, HI)
   END;  (* SLAVESORT *)
```

```
PROCEDURE MASTERSORT (VAR E : LIST; LO, HI : INTEGER);

   VAR
      MID : INTEGER;

   BEGIN (* MASTERSORT *)
      IF LO < HI THEN
         BEGIN
            IF (HI-LO < LARGE) AND (HI-LO > SMALL) THEN
               SLAVESORT (E, LO, HI)
            ELSE
               BEGIN
                  PARTITION (E, LO, HI+1, MID);
                  MASTERSORT (E, LO, MID-1);
                  MASTERSORT (E, MID+1, HI)
               END
         END
   END;  (* MASTERSORT *)

BEGIN (* QUICKSORT2 *)
   READ (NUM, LARGE, SMALL);
   FOR J := 1 TO NUM DO
      READ (E [J]);
   E [NUM+1] := MAXINT;
   MASTERSORT (E, 1, NUM);
   WAIT (SLAVESORT);
   FOR J := 1 TO NUM DO
      WRITE (E [J])
END.  (* QUICKSORT2 *)
```

APPENDIX 2.3

```
(*   This is the master module produced by the translator
for the Pascal-C version of quicksort in appendix 2.2. *)


PROGRAM QUICKSORT2 (INPUT, OUTPUT);
VAR ZZRTSBUF:ARRAY [1..1500] OF INTEGER;(*FOR PASCAL-C RTS*)


   TYPE
      ELEMENT = INTEGER;
      LIST   = ARRAY [1 .. 1000] OF ELEMENT;

   VAR
      E : LIST;
      NUM, J : INTEGER;
      LARGE, SMALL : INTEGER;

   PROCEDURE PARTITION (VAR E : LIST; LO, HI : INTEGER;
                        VAR MID : INTEGER);

      VAR
         KEY, TEMP : ELEMENT;
         K : INTEGER;


PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);
PROCEDURE RTS2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);EXTERNAL
BEGIN RTS2(ZZACTION,ZZDATA)END;
BEGIN (* PARTITION *)
         KEY := E [LO];
         K   := LO;
         REPEAT
            REPEAT
               LO := LO+1;
            UNTIL E [LO] >= KEY;
            REPEAT
               HI := HI-1;
            UNTIL E [HI] <= KEY;
            IF LO < HI THEN
               BEGIN
                  TEMP   := E [LO];
                  E [LO] := E [HI];
                  E [HI] := TEMP
               END
         UNTIL LO >= HI;
         E [K]  := E [HI];
         E [HI] := KEY;
         MID    := HI
```

```
        ;ZZPRO1(35,LO); (*CHECK FOR ACTUAL VAR PARAMS TO DOWN
                          PROCS*)
END;  (* PARTITION *)

    (*PASCAL-C RTS DEFINITIONS*)
PROCEDURE RTS1 (ACTION:INTEGER); EXTERNAL;
PROCEDURE RTS2 (ACTION,DATA:INTEGER); EXTERNAL;
PROCEDURE RTS3 (ACTION,DATA1,DATA2:INTEGER); EXTERNAL;
                SIZE:INTEGER);EXTERNAL;
FUNCTION RTS4 (ACTION,DATA:INTEGER):BOOLEAN; EXTERNAL;
(*END RTS DEFINITIONS*)

 PROCEDURE SLAVESORT (VAR E : LIST; LO, HI : INTEGER);
(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:LIST;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:LIST;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
(*END RTS REDEFINITIONS*)

BEGIN
IF RTS4(96,1)THEN BEGIN RTS3(66,1,1);(*NEW ACTIVATION UNLESS
                                       TERMINATED*)
ZZPRO1(65,E,2000); (*SEND VAR PARAMETER TO SLAVE*)
ZZPRO2(64,LO,2); (*SEND COPY OF VARIABLE TO SLAVE*)
ZZPRO3(64,HI,2); (*SEND COPY OF VARIABLE TO SLAVE*)
RTS1(16)END; (*END DOWNLOADING*)
END;  (* SLAVESORT *)

   PROCEDURE MASTERSORT (VAR E : LIST; LO, HI : INTEGER);

      VAR
         MID : INTEGER;

PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);
PROCEDURE RTS2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER);EXTERNAL
BEGIN RTS2(ZZACTION,ZZDATA)END;
BEGIN (* MASTERSORT *)
```

```
          IF LO < HI THEN
             BEGIN
                IF (HI-LO < LARGE) AND (HI-LO > SMALL) THEN
                   BEGIN RTS2(34,1);RTS1(0); (*RESERVE SLAVE
                                          & LOCK BEFORE DOWN CALL*)
SLAVESORT (E, LO, HI);RTS1(1)END (*UNLOCK AT END OF DOWN
                                          CALL*)

                ELSE
                   BEGIN
                      PARTITION (E, LO, HI+1, MID);
                      MASTERSORT (E, LO, MID-1);
                      MASTERSORT (E, MID+1, HI)
                   END
             END
      ;ZZPRO1(35,LO); (*CHECK FOR ACTUAL VAR PARAMS TO DOWN
                               PROCS*)
END;  (* MASTERSORT *)

   BEGIN (*MASTER MAINLINE*)
RTS2(48,1); (*INITIALIZE MASTER RTS*)
 (* QUICKSORT2 *)
      READ (NUM, LARGE, SMALL);
      FOR J := 1 TO NUM DO
         READ (E [J]);
      E [NUM+1] := MAXINT;
      MASTERSORT (E, 1, NUM);
      RTS2(32,1) (*WAIT (SLAVESORT)*)
;
      FOR J := 1 TO NUM DO
         WRITE (E [J])
   ;RTS1(17) (* END PROGRAM *)
END.
```

APPENDIX 2.4

(* This is the main module for the slave, created by the translator for the Pascal-C version of quicksort. *)

```
VAR RTS:ARRAY [1..1500] OF INTEGER;(*FOR PASCAL-C RTS*)
    DOWNID:INTEGER;
PROCEDURE DP1;EXTERNAL;
PROCEDURE ZZSTAT;EXTERNAL;
PROCEDURE RTS2(X,Y:INTEGER);EXTERNAL;
PROCEDURE RTS6(X:INTEGER;VAR Y:INTEGER);EXTERNAL;
BEGIN
 RTS2(48,0); (* INITIALIZE SLAVE RTS *)
 ZZSTAT; (* SAVE STATE TO RESTORE AFTER TERMINATION *)
 REPEAT
  RTS6(160,DOWNID); (* WAIT FOR DOWN PROC REQUEST *)
  CASE DOWNID OF
   1:DP1;
   OTHERWISE
  END;
 UNTIL DOWNID = -1
END.
```

APPENDIX 2.5

(* This is the external module for DOWN procedure SLAVETRY in the Pascal-C version of quicksort. *)

```
(*$E*)PROCEDURE DP1;

TYPE
LIST  = ARRAY [1 .. 1000] OF ELEMENT;
PROCEDURE
PARTITION (VAR E : LIST; LO, HI : INTEGER;
           VAR MID : INTEGER);

      VAR
         KEY, TEMP : ELEMENT;
         K : INTEGER;

      BEGIN (* PARTITION *)
         KEY := E [LO];
         K   := LO;
         REPEAT
            REPEAT
               LO := LO+1;
            UNTIL E [LO] >= KEY;
            REPEAT
               HI := HI-1;
            UNTIL E [HI] <= KEY;
            IF LO < HI THEN
               BEGIN
                  TEMP   := E [LO];
                  E [LO] := E [HI];
                  E [HI] := TEMP
               END
         UNTIL LO >= HI;
         E [K]  := E [HI];
         E [HI] := KEY;
         MID    := HI
      END;
(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
(*END RTS REDEFINITIONS*)


PROCEDURE DOWN;
(*BEGIN DECLARATIONS FOR DOWN PROCEDURE PARAMETERS*)
VAR E,ZZBUF1:LIST;(*VAR PARAM & COPY FOR UPDATE CHECK*)
ZZ1X1:1..1000; (*INDEX TO SCAN FOR UPDATES IN ABOVE VAR
                 PARAMETER*)
VAR LO:INTEGER;(*VALUE PARAMETER*)
VAR HI:INTEGER;(*VALUE PARAMETER*)
```

```
(*END OF DECLARATIONS FOR DOWN PROCEDURE PARAMETERS*)

      PROCEDURE SORT (VAR E : LIST; LO, HI : INTEGER);

         VAR
            MID : INTEGER;

         BEGIN (* SORT *)
            IF LO < HI THEN
               BEGIN
                  PARTITION (E, LO, HI+1, MID);
                  SORT (E, LO, MID-1);
                  SORT (E, MID+1, HI)
               END
         END;  (* SORT *)

(*PASCAL-C RTS DEFINITIONS*)
PROCEDURE RTS1(ACTION:INTEGER);EXTERNAL;
PROCEDURE RTS3(ACTION:INTEGER;CHANGED:BOOLEAN;
               SIZE:INTEGER);EXTERNAL;
(*END RTS DEFINITIONS*)

(*RTS REDEFINITIONS TO DEFEAT TYPE CHECKING*)
PROCEDURE ZZPRO1(ZZACTION:INTEGER;VAR ZZDATA:LIST;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:LIST;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO2(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
PROCEDURE ZZPRO3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
                 ZZSIZE:INTEGER);
PROCEDURE RTS3(ZZACTION:INTEGER;VAR ZZDATA:INTEGER;
               ZZSIZE:INTEGER);EXTERNAL;
BEGIN RTS3(ZZACTION,ZZDATA,ZZSIZE)END;
(*END RTS REDEFINITIONS*)

BEGIN (*DOWN PROCEDURE CORE*)
ZZPRO1(75,E,2000); (*RECEIVE VAR PARAMETER FROM MASTER*)
ZZBUF1:=E; (*SAVE COPY OF VAR PARAMETER TO DETECT CHANGES*)
ZZPRO2(74,LO,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
ZZPRO3(74,HI,2); (*RECEIVE COPY OF VARIABLE FROM MASTER*)
 (* SLAVESORT *)
        SORT (E, LO, HI)
```

```
ZZPRO1(72,E,2000); (*BEGIN UPDATE OF VAR PARAMETER*)
FOR ZZ1X1:=1 TO 1000 DO BEGIN
RTS3(73,E[ZZ1X1]<>ZZBUF1[ZZ1X1],2);(*CHECK FOR UPDATE*)
END;(*FOR ZZ1X1*)
RTS1(17)END; (* END DOWN PROCEDURE CORE *)

BEGIN (*REGION FOR COPY SECTION ITEMS*)
DOWN END; (*END COPY SECTION REGION*)
```