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

An Implementation Of Algol W

Simon Redding

A Major Technical Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science
Concordia University
Montéal, Québec, Canada

May 1985

ISBN 0-315-30632-7

ABSTRACT

An Implementation of Algol W

Simon Redding

This report describes a VAX implementation of the programming language Algol W using the technique of attribute grammars. The report consists of two parts. The first part describes the attribute grammars and the compiler used to translate the source program to the intermediate language Janus. The second part describes the translation from Janus to assembly code and the extensions made to Janus to make it more suitable for compiling Algol W.

Acknowledgments

I would like to thank my advisor, Dr. H.J. Boom, for his advice and assistance during the course of this project. I also want to express my appreciation for his providing the tools, parser generator and attribute evaluator, without which this project would not have been possible.

TABLE OF CONTENTS

1	Attribute Grammars	1
	1.1 Attribute Evaluation	3
	1.2 Pre-specified Traversal Stratagies	4
	1.3 Parser Generator	5
2	Algol W Implementation	6
	2.1 Compiler Overview	7
	2.2 Declarations	9
	2.3 Standard Prelude	9
	2.4 Postfix Translation	10
	2.5 Pass II	11
	2.6 Internal Representation	11
	2.7 Declaration Processing	12
	2.8 Expression Translation	13
	2.9 Parameter Passing	14
	2.10 Global and External Procedures	16
	2.11 Input Output System	17
	2.12 Optimizations	18
	2.13 Conclusions	18
3	The Janus Implementation	20
	3.1 The Janus Machine	20
	3.2 Janus Extensions	21

3.3 Garbage Collection 21

3.4 Compiler Structure 23

3.5 Lexical Analysis 24

3.6 Symbol Table 25

3.7 Parsing 25

3.8 Code Generation 26

3.9 Operand Addressing 26

3.10 Activation Records 28

3.11 Parameter Passing 29

3.12 Collectable Records 29

3.13 Data Representation 30

3.14 Conclusions 31

References 32

Appendix 1: Algol W to Janus Translation 33

Appendix 2: Janus to Assembly Translation 34

1 Attribute Grammars

The most widely used method to describe programming languages in the past has been the use of context-free grammars(CFG) [Aho77]. Although they are sufficient to describe the context free syntax, most programming languages have context-sensitive syntax such as the relationship that must hold between the declaring and applied occurrences of identifiers. Such constraints are typically imposed using a set of rules written in English.

An alternative approach is to define the language using an attribute grammar[Waite84]. Attribute grammars extend the power of CFG's by allowing context-sensitive constraints to be defined which specify the relationship between a phrase and its context.

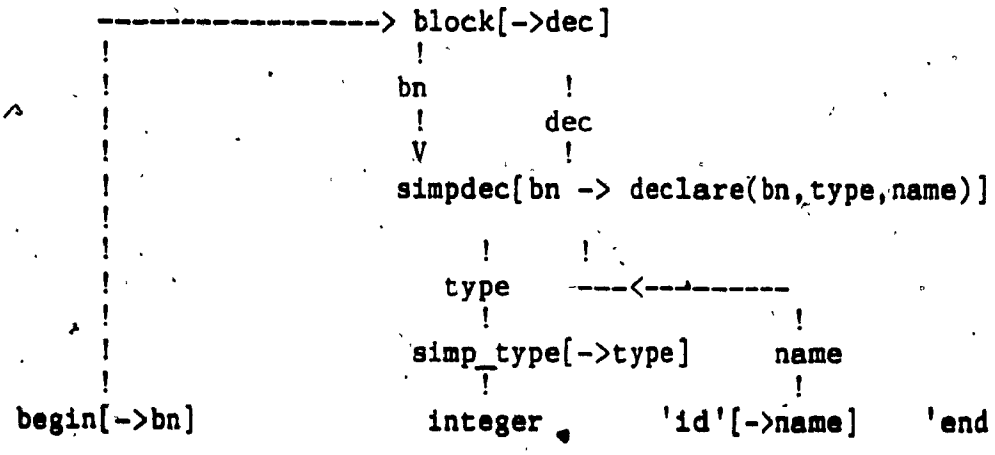
An attribute grammar is a context free grammar which has been extended to define the context sensitive constraints. Each symbol in the grammar has associated with it a set of attributes which represent attributes of the language construct. Each production in the grammar may contain an attribute function which specifies the relationship between the attributes and thus only the values consistent with the context-sensitive constraints will be allowed. The attribute functions take the place of the semantic functions in a conventional compiler which check the consistency of the synthesized attributes with their environment. The advantage is that the semantic functions are specified in the grammar and not separately in the body of the compiler.

Consider the grammar for a simple declaration:

```
block[-> dec] :  'begin'[->bn], simpdec[bn -> dec], "end".

simp_dec[bn -> declare(bn,type,name)] : simp_type[ -> type],
                                        id[-> name].
```

In these two productions the left and right sides are seperated by a colon. The symbols enclosed in square brackets are the attributes for the productions. Attributes on the left side of the arrow are inherited attributes and those on the right are derived or synthesized attributes. Inherited attributes on the left side and derived attributes on the right side of the production are the defining occurrences, that is their values are passed into the production. The derived attributes on the left and inherited attributes on the right are called applied occurrences and take their values from the defining attributes. The following is a parse tree for the block declaration, with the direction of attribute flow shown by the arrows.



In the above declaration 'bn' is the block number for the declaration and is inherited from the context. The type and name are derived from the

phrase itself. The attribute function **declare** uses the values of the attributes, bn, type and name, to declare the variable. Thus declare is an applied attribute and the others are defining attributes. Derived attributes move up the tree while inherited attributes move down the tree.

The reason for using attribute grammars is not only to combine both the syntax and semantics into one formalism but also to be able to automatically generate a compiler from the language specification. The attributes used to define a translation of a language will be such things as data types, symbol tables, operator types and instructions representing the translation.

1.1 Attribute Evaluation

The evaluation of applied attributes can only be carried out when the values for all of the defining attributes, on which the applied occurrences depend, are known. Thus if the block number were not known until the end of the block, in which the declaration is made, then the evaluation of the declaration would have to wait until the entire block had been scanned. One method is to build the entire parse tree and then recursively move up and down the tree attempting to evaluate attributes. If all the dependencies for a particular attribute function are satisfied in a production then the attribute function can be evaluated and the applied attribute set to the computed value. This is continued until all the attributes have been evaluated. If it is possible for the value of an attribute to depend on itself then the grammar has a circularity and the sentence which causes this cycle cannot be evaluated using the grammar.

1.2 Pre-Specified Traversal Strategies

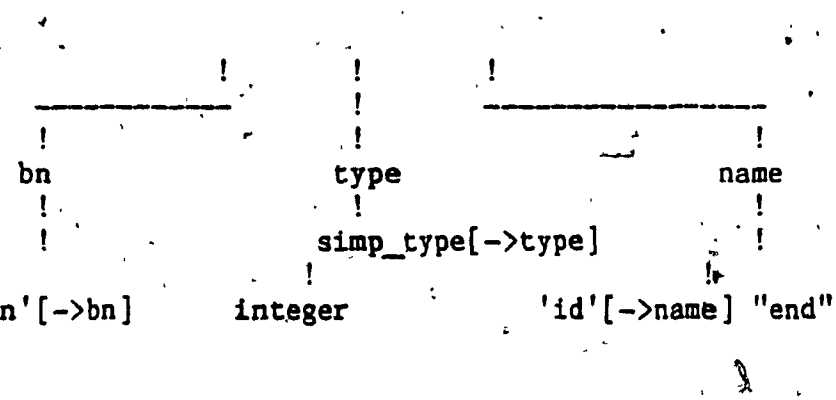
Given an acyclic attribute grammar the only constraint on evaluation order of attribute functions is that the dependencies must be satisfied. Evaluation of an attribute function, with unsatisfied dependencies, can be avoided if an evaluation order is specified, which never attempts to evalute an attribute before the dependencies are satisfied. In this way there will be no wasting of time by attempting to evaluate a function, failing, and then having to store the necessary information to complete the evaluation later.

One possible strategy is to evaluate the functions just before a reduction is made, in a bottom up parser. To insure that all the defining attributes are known at this time, it is sufficient, but not necessary, to have a grammar which consists of only derived attributes. Since all the information is contained in the phrase about to be reduced it is assured that the attributes will have values. It is obvious that the grammar is acyclic because the attribute flow is always upward in a tree, which is an acyclic graph. Thus a grammar consisting of only derived attributes is guaranteed to have the property that it can be evaluated in a single left to right pass.

It is always possible to remove inherited attributes from a grammar by a method known as hoisting. This technique moves attributes up the parse tree until the inherited attributes, on which the applied attributes depend, become derived attributes. At this point the applied attribute which depended on the inherited attribute can be evaluated.

Thus the attribute function for a declaration could be moved up to the production for a block at which point the block number would be a derived attribute:

```
block[->declare(bn,type,name)] : 'begin'[->bn],
                                 simp_dec[->type,name],
                                 "end".
```



Thus, just before the above reduction is to be made, the values of all the defining attributes used in the **declare** function are guaranteed to be known, because the terminals and nonterminals for the production are now on the parse stack. In this example 'begin' is a terminal and the value of its attribute is supplied by the lexical analyzer by keeping track of the number of begins and ends already seen.

1.3 Parser Generator

The parser generator used takes as its input the specification of a programming language in a BNF extended with attributes and attribute functions. The output is a set of LR(1) parsing tables, along with instructions for the attribute evaluator. The instructions, for attribute evaluation, are responsible for moving attributes around the parse tree

and calling attribute functions. Thus the parser generator automatically constructs a program, which when given a correct sentence, can parse it and determine its meaning.

When an attribute function is about to be called the interpreter will check that all of the arguments are known and, if they are, it calls the appropriate function with them. If there are one or more unknown arguments then the interpreter cannot call the function but will have to wait until they are known. This involves storing values on a dependency list. When an atrribute on a dependency list becomes known, then another attempt is made to evaluate the attribute function. Thus the interpreter may make many attempts at evaluating an attribute function before it succeeds. This is the reason a pre-specified evaluation order, which guarentees that the values are known, is important.

2 Algol W Implementation

The Algol W translator transforms the source text into the intermediate language Janus in two passes. A least two passes are needed because declarations are allowed to follow applied occurrences.

The translation could accomplished by writing a grammar and specifying that the attributes be calculated in two passes over the structure tree. The first pass would evaluate the declarations which would then be used by the second pass.

It was decided, for reasons of simplicity, to use two grammars and to write the result of the first pass out to file store and have pass 2 read

it in and process it with another grammar. The attributes calculated in the first pass are represented by a symbol table.

The first pass processes declarations and transforms expressions into postfix form. The second pass, using the symbol table built in the first pass, translates the postfix code to Janus code.

The grammars for both passes use only derived attributes. The reason for this is that it is obvious that all the dependency constraints will be satisfied before an attribute function is invoked and that the grammar is acyclic.

If an attempt to evaluate an attribute function fails because one of the attributes is not known, this is flagged as a compiler error since the grammar has been designed so that this cannot occur.

2.1 Compiler Overview

The compiler consists of five modules each of which is responsible for a particular function.

The modules are:

1) OUT.PAS	responsible for output
2) STRING.PAS	string manipulation
3) VALUES.PAS	data structures and reference counting
4) CALL.PAS	attribute function evaluator
5) PARSE.PAS	lexical analysis and parse table interpreter and control

The first pass is responsible for processing declarations and transforming expressions to postfix form.

The files used by the first pass are:

1)	TABLES.DAT	the parse tables and attribute evaluation tables
2)	STPREE.DAT	the standard prelude containing pre-declared routines
3)	PAS$INPUT	the source program to be parsed
4)	PAS$OUTPUT	the listing file and error messages
5)	OBJECT.DAT	the postfix translation
6)	SYMFILE.DAT	the symbol table
7)	BLOCKTAB.DAT	the block nesting table

The attribute grammar for this pass describes the context-free syntax of Algol W and specifies the translation to postfix code. The two main attributes which are moved around are the translation to postfix and the symbol table.

A block nesting table is also constructed which gives the block number of the surrounding block for each block in the program. Whenever a block is entered an attribute function is called to give the block number. As a side effect, this function call stacks the current block number and enters the block number and its surrounding block number in the block table. On leaving a block the surrounding block number replaces the current block number.

2.2 Declarations

As declarations are processed, entries are made in the symbol table and, in some cases, a string is generated for the second pass to read.

The information, entered in the symbol table, includes the name of the object being declared, the block number, a unique number used for identifier generation, and in the case of procedures, records and references other information concerning parameters, fields and record classes.

Simple variable declarations result in entries being made in the symbol table but do not produce any translated string.

Array and procedure declarations not only enter information in the symbol table but also result in a translated string. Array declarations must be carried over into the second pass because they can contain expressions which cannot be evaluated at compile time. Thus array declarations are executable. Only the number of dimensions and the type of element is known at compile time.

For procedure declarations the body of the procedure must be passed on but the formal parameter list is be stored in the symbol table.

2.3 Standard Prelude

The entire program is enclosed by a fictitious block, bearing number zero. This block contains all the pre-defined functions such as sin, cos, trunc amd round. The declarations for the standard prelude are actually in a file called STPREE.DAT which is read before processing the source text.

The postfix translation and the symbol table are the two attributes of interest in the first pass. The partial translation and symbol table are derived attributes of all nonterminals. When a reduction is performed the tables, which contain the declarations for the production being reduced, are merged and the strings, resulting from the translation, are concatenated. The translation process terminates when the finale reduction to the distinguished nonterminal PROGRAM[->str,env] is performed. At this point the postfix translation, the symbol table and the block nesting table are written out to file store.

Since the first pass is not concerned with the meaning of the program to be translated, different features of the language which have identical syntactic descriptions can all be handled by a single grammar rule. For instance array references and procedure calls with parameters are treated identically by the translator. The distinction will be made on the second pass with the aid of the symbol table.

2.4 Postfix Translation

The translation of expressions from infix to postfix is performed whenever a reduction is performed involving infix operators.

```
term[->combine(env1,env2), conc(str1,str2,"div")] :
            term[->env1,str1], "div",
            factor[->env2,str2].
```

The environments, **env1** and **env2**, which represent declarations, are combined and the strings representing the term and factor are concatenated followed by the operator div.

Thus the expression

A + B * C

would become

A B C * +

Control structures such as **if then else, case statement** and **goto** statements are not transformed and appear in the same form in the translation as in the source.

2.5 Pass II

This pass uses the symbol table and block table built in the first pass to translate the postfix string to the intermediate language Janus. The files used in this pass are:

1) TABLES2.DAT — the parse tables and attribute tables
2) SYMFILE.DAT — the symbol table from the first pass
3) OBJECT.DAT — the postfix translation from the first pass and the output of the second pass
4) PAS$OUTPUT — any error messages are appended to this file
5) BLOCKTAB.DAT — block nesting table

2.6 Internal Representation

The most important attribute is the one which represents the object code. Since this attribute is a derived attribute of the start symbol it is important to find an efficient representation in terms of space. Most of the string will be made up of keywords with less than 30% being other than Janus reserved words. To avoid storing the same string many

times each keyword is stored only once and a pointer to it is used in its place in the translation.

As in the first pass, all the attributes are derived and the translation is formed by concatenating the translations together. The attribute functions construct a translation, given as input derived attributes which contain the translations of the sub-phrases and the typing information, and constructing the translation. Unlike the first pass, in which the translation does not re-order nonterminals, the second pass may re-order the nonterminals in the translation. Thus it is not possible to emit the translation at every reduction but it is neccesarry to store arbitrarily large sections of the translation until all remaining translations involving the sub-phrases are simple postfix. The second pass starts by reading in the symbol and block tables, from the file store, and storing them in the named data structures **symbol_table** and **block_table** respectively. Record class declarations are also stored in a list called **definition_table** when the symbol table is read in. This is used to produce all the record mode definitions needed in by the Janus translator.

2.7 Declaration Processing

The second pass then proceeds to read the translation resulting from the first pass. When a block is entered an attribute function is called, with the block number as an argument, which will find all the variable declarations in the block just entered and produce a string consisting of Janus declarations for them. If the block entered was a procedure

declaration then declarations for **result** and **value result** parameters are produced along with code to load the variable with the initial value in the case of **value result** parameters.

Array declarations are emitted when the translated string for them is encountered. A dope vector for them is declared which consists of a base pointer and as many lower bound stride pairs as there are dimensions. Code is then emitted which initializes the dope vector, grabs a section of the stack sufficient for the array, and sets the base pointer to point to it.

2.8 Expression Translation

The translation from postfix code to Janus proceeds by simulating the execution of the program on a stack machine and producing the Janus instructions to push operands onto a stack and then operating on them. The resulting attributes from the reduction are the string containing Janus code that performs the operation and an attribute which describes the result. In this way the type of the intermediate result is known to any other productions which use it.

```
expression[->apply(op,arg1,arg2,type1,type2),
                   coerce(type1,type2,op)] :
                   variable[->arg1,type1],
                   variable[->arg2,type2],
                   operator[->op].
```

The attributes **type1** and **type2** give the type of the operands including whether they are on the stack or must be put there. If an argument is a simple variable then the type will indicate this and will contain all the information needed to reference it. If an arguments is

not simple operand but is an intermediate result then arg1 or arg2 will be a string which is the translation to calculate its value and put it on the stack. If it is a simple variable then the string will be empty. The function apply will take the arguments and builds a translation for the entire operation. The function coerce will use its parameters to determine the type of the result and return an attribute with information describing it.

The translation of A B C * + , assumuing A, B and C are integers, would be:

```
load int disp n a1.
load int disp n a2.
load int disp n a3.
mpy int.
add int.
```

Here **a1**, **a2** and **a3** are compiler generated unique identifiers for A, B and C respectively. The **disp n** indicates the display level for the variables, with 0 for the outermost level.

The attribute function coerce would return an attribute indicating that the result type is integer and that it is on the stack. If the operands had been of different types then the apply function would have placed the appropriate conversions between the operands and the coercion would have returned a value consistent with the result.

2.9 Parameter Passing

Algol W supports six different parameter passing mechanisms. They are **value**, **value result**, **result**, **name**, **procedure** and **formal array**. In addition to these a seventh has been added, which is pass by reference,

For **result** and **value result** paramaters the address of the parameter is passed to the called procedure. The called procedure uses this address to modify the parameter at the end of the call and in the case of **value result** it initializes a local variable with the value of the actual parameter.

Formal array parameters have not been fully implemented. They can only accept entire arrays of the same rank as the declaration and not subarrays. They are implemented by passing the address of the dope vector and the base address which are then copied to a local versions. All references to the actual parameter are then through this new dope vector to the original array.

Name parameters require the address of the variable at the time of reference, not at the time it was passed, which, in the case of array elements, may change between successive accesses. To implement this, a procedure called a thunk has to be constructed which will return the address of the variable. The entry point to the thunk is passed to the called procedure instead of the address of the variable. The called procedure calls the thunk whenever it needs to know the address.

Since **name** parameters are so expensive to use it was decided that it would be useful to implement the less expensive mechanism of passing by reference, which is wanted anyway.

Pass by **reference** simply passes the address of a variable to the

called procedure, which uses it to reference the variable.

Procedure parameters are implemented by building a procedure which has as its body the actual parameter. The reason the procedure closure for the actual paramter cannot be used is that the actual parameter may not be a procedure but a statement. For instance the following program is valid:

```
begin
procedure foo(procedure p1);p1;
foo(   begin
         integer a1,a2;
         writeon(a1,a2);
       end.)
end.
```

Thus a new procedure must be built which has the same effect as executing the statement and its closure passed as the actual parameter.

2.10 Global and External Procedures

The Algol W implementation allows for procedures to be compiled seperatly and, the object modules produced, linked together. The syntax of an **external/global** procedure declaration is:

procedure_heading: "global"!"external", [procedure_body].

Procedure body is not present if it is a external declaration as it is defined elsewhere.

The procedure heading in the declaration gives the name of the procedure and the types of the formal parameters.

If it is a Pascal procedure which is being linked then it is important

that the correct formal types be used as Pascal does not support all the passing mechanisms of Algol W.

Pascal usually expects the address of the variable to be passed unless the formal declaration uses the attribute %immed, in which case call by value is assumed. If real parameters are to be passed they must always be either **call by reference**, **call by result** or **call by value result**. The reason call by value is not allowed is that %immed parameters are not allowed to exceed 32 bits by VAX Pascal.

2.11 Input Output System

Janus does not define any I/O routines so these must be supplied by external procedure calls.

The routines are to be used as if they were declared with the following procedure headings:

```
# RESULT LEFT ON THE STACK.
BEGIN INT DISP 0 GETINT_$.
PAREND INT DISP  0 GETINT_$.

#RESULT LEFT ON THE STACK.
BEGIN REAL DISP 0 GETREAL_$.
PAREND REAL DISP 0 GETREAL_$.

#  STRING OF LENGTH P1 READ INTO  THE  LOCATION P2.
BEGIN DISP 0 GETSTR_$.
PARAM INT DISP 0 P1.
PARAM ADDR DISP 0 P2.
PAREND DISP 0 GETSTR_$.
# P1 IS OUTPUT.
BEGIN DISP 0 PUTINT_$.
PARAM INT DISP 0 P1.
PAREND DISP 0 PUTINT_$.
```

```
# P1 IS OUTPUT.
BEGIN DISP 0 PUTREAL_$.
PARAM REAL DISP 0 P1.
PAREND DISP 0 PUTREAL_$.
# STRING STORED AT P1 OF LENGTH P1 IS OUTPUT.
BEGIN DISP 0 PUTSTR_$.
PARAM INT DISP 0 P1.
PARAM ADDR DISP 0 P2.
PAREND DISP 0 PUTSTR_$.
```

2.12 Optimizations

At present the translator is rather slow and some rather simple optimizations may improve its performance considerably.

The translator takes about 32 CPU seconds to perform the initializations for both passes. If the initializations were embedded in the compiler in the form of constant declarations then no time would have to be spent doing initalizations.

In the first pass it would certainly be possible to emit the translation immediately upon reduction because the translation scheme is simple postfix. That is the order of the non-terminals in the translation is the same as in the grammar. Thus instead of concatenating strings the translator could call an output function to emit the translation.

2.13 Conclusions

The compiler described uses only attribute grammars to transform the source text into intermediate code. This is unusual. Although attribute grammars have been used in the past they are not usually used in the synthesis of object code .

Because the translation is specified by the grammar it was found that

many changes that had to be made to correct errors could be made in the grammar with no need to recompile the rest of the compiler. This resulted in a much faster turnaround time in the development than otherwise would have been the case with a conventional design.

3 Janus Implementation

This part of the report describes the implementation of a compiler for the intermediate language Janus. It describes the structure of the compiler and the files used by it. The extensions made to Janus, and their implementation, are also described.

3.1 The Janus Machine

The Janus machine presents a low level machine in which instructions operate on primitive data types but also allows features useful for implementing high level languages such as procedure calls, array referencing, record structures and address manipulation[Waite75]. The language is sufficiently restricted so as to permit an efficient implementation and yet sophisticated enough to permit an easy transformation from high level languages.

Central to the Janus machine is the operand stack. This structure may hold operands or intermediate results and is the destination of arithmetic operations. Since Janus was designed to be implemented efficiently on many machines, not only those with stacks, it has certain properties which enable the translator to know the depth of the stack at any time. Thus it is possible to implement the operand stack using registers. To ensure that the stack depth is always known, the stack is emptied on block entry and exit and at any occurence of a label. Thus it is not possible to have the stack grow indefinitly by looping with a jump statement.

3.2 Janus Extensions

Janus has been extended with three features to simplify translation from Algol W.

FREEZE blocks were introduced to overcome the problem of having the operand stack empty every time a label is encountered. Without the ability to keep the operand stack intact after a branch the implementation of conditional expressions requires an inefficient sequence of stores and loads. To overcome this problem a new construct was introduced, similar to blocks except that the operand stack is marked instead of being voided on entry. In addition an, explicit **break** command is allowed which causes control to transfer to the instruction following THAW instruction. When the block is left, either with a break or by falling through the **THAW**, the mark is moved back to the position it had on entering the **FREEZE block**. The syntax of **FREEZE** is the following:

FREEZE symbol

(code element eol)*

THAW symbol

3.3 Garbage Collection

Algol W requires that records be stored on a heap and does not require the programmer to explicitly free them when he no longer needs them. To prevent the system from making unreasonable storage demands some means of re-using storage, which has become inaccessible, must be provided. Thus

the problem of garbage collection arises.

Janus provides a heap for storage which does not obey a stack discipline. It also provides grab and free operations which will get blocks of memory and return them to the free storage pool. A garbage collector could be written which manages the heap and calls the free function at the approprite times.

Instead of doing this it was decided to augment the Janus machine with features which would free the programmer from the need to write his own garbage collector and instead use built-in features of Janus.

To do this it was necessary to introduce a new data type and instructions to manipulate them.

The new data types are pointers to records which will be allocated on the heap. A new record mode definition was also introduced which informs the translator that the record is going to be allocated space on the heap the syntax of which is:

```
COLLREC SYMBOL EOL
    record layout
COLLEND SYMBOL EOL
```

This defines SYMBOL as a record mode which is to be allocated on the heap.

The pointers are declared using the new primitive type COLLECT, which tells the translator that it points to something on the heap so the garbage collector knows where to find it.

To create a record on the heap the **ALLOC** statement is executed which specifies the record mode and the collect mode object which is to point to it. The generated code from this statement will check that space of sufficient size for the record is allocated and the tag field in the record is set to the appropriate record class. If the collect mode object is already pointing at a record on the heap then the translator checks that it is of the correct class and generates an error condition otherwise.

At present storage is not reclaimed when the heap storage is exhausted but s system call is made to get another block of memory.

The syntax of Janus has been extended to allow procedure declarations to be properly nested. This was done because it is necessary to construct procedures when processing procedure parameters which are either call by name or procedure. Thus it is not necessary for the high level language translator to unravel the block structure.

3.4 Compiler Structure

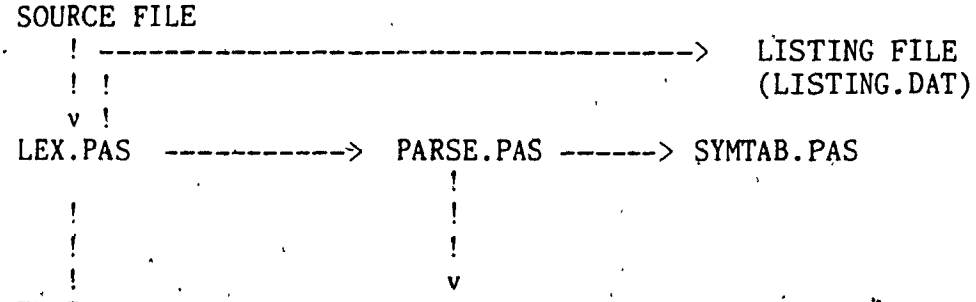
The compiler uses a recursive descent parser written in Pascal. It takes as its input Janus code and produces VAX assembly code suitable for input to an assembler.

The compiler consists of four modules which are:

	Name	Function
1)	SYMTAB.PAS	symbol table managment
2)	CODE.PAS	code generation

3) LEX.PAS lexical analysis

4) PARSE.PAS recursive descent parser



3.5 Lexical Analysis

LEX.PAS is responsible for reading the source text and providing the parser with tokens. The lexical analyzer is coded as a finite state machine, which models the lexical structure of Janus.

The lexical analyzer is called by the parsing routine whenever it needs another token. The lexical analyzer then scans the input and gets the next token. Depending on the type of token different things are returned to the parser. The following are the possible categories of token:

1) Janus keyword including '+', '-', '*', '(', ')', '[', ']' and '.'

2) Literal constant (integer , real or character)

3) Identifier

If it is not obvious that a token is not a keyword then the lexical analyzer searchs an array, built at initalization time from KEYWORDS.DAT, of reserved words and if it is found then the code for the

keyword is returned. If the token is not found then the token must be an identifier and the character string representing it is returned.

The lexical analyzer is also responsible for producing a listing file, LISTING.DAT.

3.6 Symbol Table

The task of storing and retrieving information about programmer declared objects is handled by the module **SYMTAB.PAS**. The only interface to this module is through the two routines **DECLARE** and **LOOKUPNAME**.

Because all identifiers must be distinct the symbol table can be very simple. Thus it is possible to use the identifier itself as the key into the symbol table.

DECLARE takes one parameter, a variant record desrcibing the thing to be declared and places it in the symbol table.

LOOKUPNAME will, given the name of an identifier, return a pointer into the symbol table for the entry.

3.7 Parsing

The module **PARSE.PAS** is the central controlling routine which calls the routines, in the other modules, at the appropriate times.

The parser has one routine for each nonterminal, N, in the grammar which parses the phrase for N by matching the terminals in the production with those in the input and then calling the appropriate routines for the

subphrases, based on the next token.

Most of the routines simply gather information which is passed back to the calling routine via parameters. After having gathered the needed information, some of the routines call a routine in the code generator with the appropriate parameters set, to emit code.

3.8 Code Generation

The code generator consists of a register allocator and procedures for constructing activation records, setting the index register, addressing operands and emitting instructions for operations. The registers are grouped according to function as follows:

R0 - R1	:	R0 passes the current addressing environment to called procedures. R0 and R1 are used to return values from functions
R2 - R3	:	Base and index registers for the Janus machine.
R4 - R11	:	General operand addressing registers. Each register can contain the address of the start of an activation record, or, in the case of procedure parameters, it can be set to the argument pointer for the procedure.
R12	:	Argument pointer.
R13	:	Frame pointer.
R14	:	Stack pointer.
R15	:	Program counter.

3.9 Operand Addressing

Operands can be stored in one of five places:

1) In the activation record for a procedure.

2) On the operand stack.

3) At a static location.

4) In parameter storage.

5) On the heap.

Depending on where an operand is located different information is needed to access it.

For an operand within an activation record and for parameter storage, the display level and offset are needed.

Heap storage is accessed using a collect modes variable as a pointer to it.

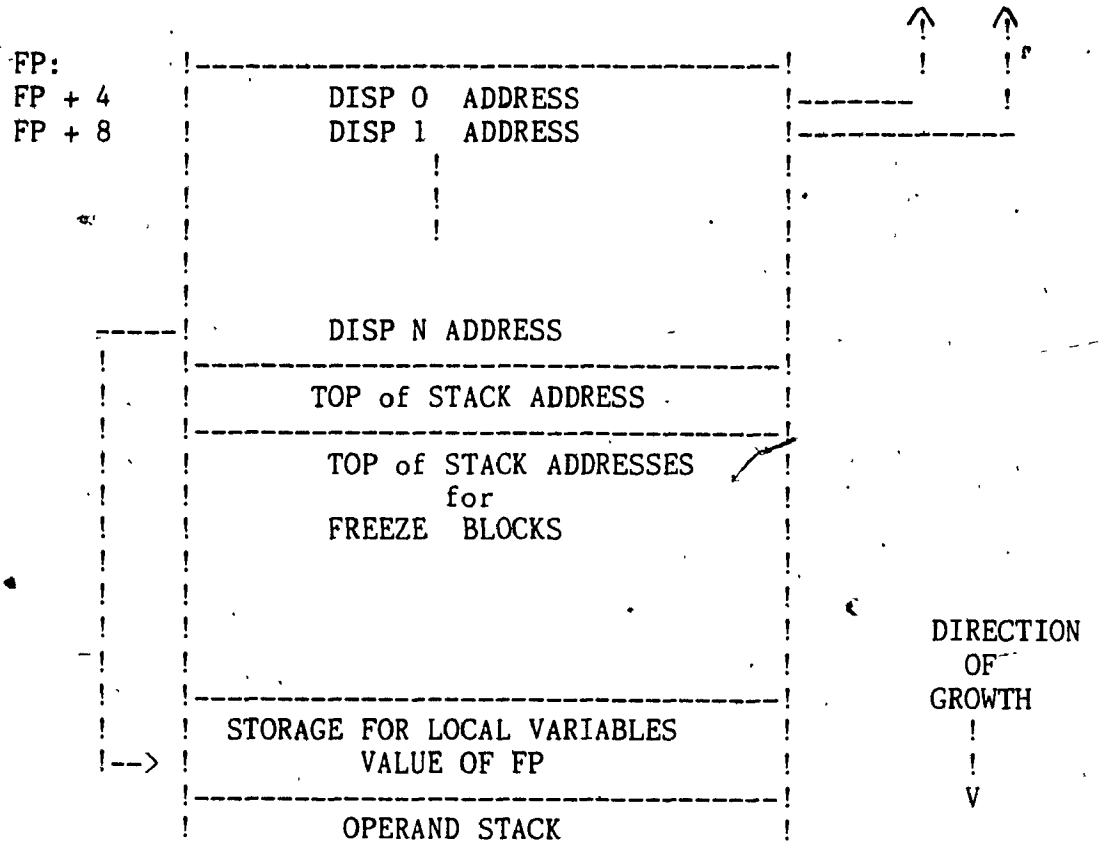
The address of static storage can be determined at assembly time and symbolic addresses are used in this case.

In addition to the above categories, storage can be based, can be indexed, and can have a field selector if it is a field of a record.

When a storage location is to be accessed the register allocator is called. If the storage category indicates the variable is either in an activation record or is a parameter then the register allocator checks to see if one of the registers R4 - R11 contains the address of the begining of the approprite data segement. If it finds that no register contains the correct address then it must emit instructions to place it there. If none of the registers are available then one is picked and its contents replaced with the address needed.

3.10 Activation Records

The activation records contain the local variables for procedures along with information to access the activation records of surrounding procedures.
The form of the activation record for DISP N is:



The activation record contains pointers to all the activation records for procedures that surround it. These are needed to reference variables which are not local to the current procedure.

The TOP of STACK ADDRESS contains the address of the top of the operand stack when it is empty. This is needed when the operand stack is voided at a label or and on block entry and exit. The value of the frame pointer is used when a jump instruction transfers control out of the current procedure and must restore the correct addressing

environment.

The TOP of STACK ADDRESSES, for the freeze blocks, are used to store the address of the top of the operand stack on entry to a FREEZE BLOCK. When the FREEZE is exited the stored TOP of STACK ADDRESS is used to replace the current TOP of STACK ADDRESS.

When a procedure is called it expects the frame pointer of the statically enclosing procedure to be in R0. Using this address, the called procedure copies as many display elements from the procedues activation record as are needed.

The **stack pointer** is then decremented by an amount sufficient to accomodate the local variables, the top of stack pointer and the freeze thaw stack tops.

3.11 Parameter Passing

All parameters for procedures are passed by pushing them onto the stack and then using the **argument pointer** to access them. When a parameter that is not local to the current procedure is to be accessed, then the static chain is searched backwards to the approprite activation record. From this the stored argument pointer is retrieved and its value placed in one of the registers R4-R11.

3.12 Collectable Records

When the Janus translator processes collectable record mode definitions it stores a record class number with its definition in the

symbol table. At the end of translating the program the symbol table is searched and all the collrec mode definitions found. With these a table is built which associates the record class with a set of offsets. The offsets are the locations, inside the record, of all collect mode objects. These are needed by the garbage collector so it can trace through the heap during garbage collection. In addition, the size of the record is stored which is used when allocating space on the heap for the record.

3.13 Data Representation

Primitive data objects occupy the following number of bytes:

BOOL and CHAR	:	One byte.
INT, COLLECT and ADDR	:	Four bytes.
REAL, PROC	:	Eight bytes.

Composite mode objects occupy space equal in size to the sum of its components. All storage is packed and the keyword **allign** has no effect on the storage allocated.

3.14 Conclusions

The Janus compiler described has a simple structure and does not attempt to perform any optimizations.

Using Janus as an intermediate language allows the front end of the compiler to deal only with the structure and semntics of the program to be compiled. The frontend makes no assupmtions about the underlying machine architecture but simply generates Janus code. The advantage to splitting the compiler into two parts, one machine-independent and the other very machine dependent, is that implementing the compiler on a different machines requires only the writing of another Janus translator.

The ease of implementation was helped by the extensive instruction set of the VAX. In most cases Janus instructions required only one or two assmbly instructions.

REFERENCES

Aho, A.V. and Ullman, J.D. [1977]
Principles of Compiler Construction.
Addison Wesley Publishing.

Barrett, W.A. and Couch, J.D. [1979]
Compiler Construction: Theory and Practice.
Science Research Associates.

Greis, D. [1971]
Compiler Construction for Digital Computers.
Wiley.

Goos, G. and Waite, W.M. [1984]
Compiler Construction.
Springer Verlag.

Kennedy, K. and Ramanathan, J. A. [1979]
Deterministic Attribute Grammar Evaluator
Based On Dynamic Sequencing.
ACM Transactions on Programming Languages and Systems, Vol 1,
No. 1, July 1979, Pages 142-160

Waite W. M. [1975]
The Universal Intermediate Language Janus.
University of Colorado.

Waite W. M. [1978]
The Universal Intermediate Language Janus.
Revised Report.
University of Colorado.

Watt, D.A. [1977]
The Parsing Problem for Affix Grammars.
Acta Informatica 8, 1977, Pages 1-20

Appendix 1 : Algol W to Janus Translation.

The following translation illustrates the use of FREEZE blocks in conditional expressions.

```
begin
  integer max , a, b ;

  max := if a > b then b else a;
end.
```

```
start algolw.
begin main ml    .
parend main ml   .
space int disp 0  al1 .
space int disp 0  al2 .
space int disp 0  al3 .
freeze f16.
load int disp 0  al2 .
load int disp 0  al3 .
rlt int .
cmp (n) bool m true .
jmp ne  r al4.
load int disp 0  al3 .
break f16.
loc al4.
load int disp 0  al2 .
break f16.
thaw f16.
store (n) int disp 0  al1 .
end main ml    .
finish algolw.
```

Appendix 2: Janus to Assembly Translation.

```
start algolw.
def int m al7 a int 20.
def int m al6 a int 10.
begin main m1 .
parend main m1 .
space int disp 0 al1 .
space int disp 0 al2 .
begin int disp 1 al5 .
param proc disp 1 pl3 .
param int disp 1 pl4 .
parend int disp 1 al5 .
rcall addr param pl3 .
rcend addr disp 1 pl3 .
base addr .
load int based .
load int param pl4 .
add int .
end int disp 1 al5 .
load int m al6 .
store (n) int disp 0 al1 .
load int m al7 .
store (n) int disp 0 al2 .
rcall r putint$ .
rcall int r al5 .
begin addr disp 1 pl8.
parend disp 1 pl8.
setloc addr disp 0 al1 .
end addr disp 1 pl8.
proc disp 1 pl8.
rarg proc .
load int disp 0 al2 .
rarg int .
rcend int r pl5 .
rarg int .
rcend r putint$ .
rcall r newline .
rcend r newline .
end main m1 .
finish algolw.
    .title  algolw
    .entry  algolw    ,^m<r4>
    brw     start
labl:
    brw     lab2
lab3:
    movl    12(r12),r0
    calls   #0,@8(r12)
    addl2   #0,sp
```

```
        pushl   r0
        movl    (sp)+,r3
        movl    0(r3),-(sp)
        movl    4(r12),-(sp)
        addl3   (sp)+,(sp)+,-(sp)
        movl    (sp)+,r0
        ret
        .entry  a15         ^m<r3,r4,r5,r6,r7,r8,r9,r10>
        subl3   #4,sp,sp
        movc3   #4,-4(r0),(sp)
        subl3   #20,sp,sp
        movl    fp,(sp)
        movl    sp,-8(fp)
        movl    sp,-12(fp)
        brw     lab3
lab2:
        movl    #10,-(sp)
        movl    -4(fp),r4
        movl    (sp)+,4(r4)
        movl    #20,-(sp)
        movl    (sp)+,8(r4)
        brw     lab4
lab5:
        movl    -4(fp),r4
;           4           0
        moval   4(r4),-(sp)
        movl    (sp)+,r0
        ret
        .entry  p18         ^m<r3,r4,r5,r6,r7,r8,r9,r10>
        subl3   #4,sp,sp
        movc3   #4,-4(r0),(sp)
        subl3   #20,sp,sp
        movl    fp,(sp)
        movl    sp,-8(fp)
        movl    sp,-12(fp)
        brw     lab5
lab4:
        movl    fp,-(sp)
        moval   p18         ,-(sp)
        movl    -4(fp),r4
        movl    8(r4),-(sp)
        movl    fp,r0
        calls   #0,a15
        addl2   #12,sp
        pushl   r0
        movl    fp,r0
        calls   #0,putint$
        addl2   #4,sp
        movl    fp,r0
        calls   #0,newline
        addl2   #0,sp
        $exit_s
        ret
start:
        calls   #0,m1
```

```
ret
.entry   m1          ^m<r3,r4,r5,r6,r7,r8,r9,r10>
subl3     #28,sp,sp
movl       fp,(sp)
movl       sp,-4(fp)
movl       sp,-8(fp)
calls     #0,initSawio
calls     #0,initgar-
brw        lab1
.end     algolw
```