

1979

PL/STAR, a structured assembly language for the CDC STAR-100

Barry Lee Mowday

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Mowday, Barry Lee, "PL/STAR, a structured assembly language for the CDC STAR-100" (1979).

Dissertations, Theses, and Masters Projects. Paper 1539625049.

<https://dx.doi.org/doi:10.21220/s2-djk9-hd77>

This Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

PL/STAR
,
A Structured Assembly Language
For The CDC STAR-100

A Thesis
Presented to
The Faculty of the Program in Applied Science
The College of William and Mary in Virginia

In Partial Fulfillment
Of the Requirements for the Degree of
Master of Science

by
Barry L. Mowday
1979

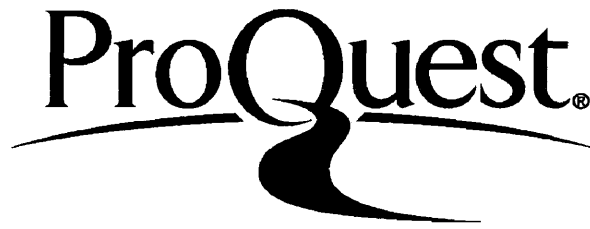
ProQuest Number: 10626208

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10626208

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

APPROVAL SHEET

This thesis is submitted in partial fulfillment of
the requirements for the degree of

Master of Science

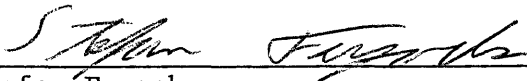


Author

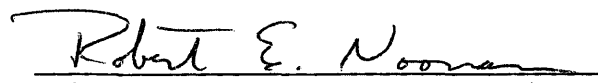
Approved, July 1979



John C. Knight



Stefan Feyock



Robert Noonan

Table of Contents

Chapter	Title	Page
	Acknowledgments	iv
	Abstract	v
I	Introduction and Basic Concepts	2
II	Language Description	9
III	Syntax	13
IV	Implementation of the PL/STAR Compiler	28
V	The Module Listing and Sample Modules	50
VI	The Compiler Writing System	64
VII	Evaluation	68
Appendix	The PL/STAR Grammar	72
	Notes	76
	References	77

ACKNOWLEDGMENTS

The author wishes to express his appreciation to Dr. John Knight for his patience and encouragement in the preparation of this work, to Douglas Dunlop for his valuable comments and to Marie Noland for her instruction in the text processing facilities used to produce this paper.

ABSTRACT

This thesis documents the programming language PL/STAR. PL/STAR is a structured assembly language for the CDC STAR-100 computer. The language is designed for systems programming applications. PL/STAR combines the control and data structures of high level languages with the access to machine features of assembly level programming. The high level features of the language supply the advantages in the development, readability and organization of programs normally associated with high level languages. The access to the registers of the machine and the control the PL/STAR programmer exercises over the machine instructions that are generated distinguish PL/STAR from higher level languages. The result is a programming language that can offer advantages over assembly language for systems programming tasks.

The compiler for PL/STAR is a syntax directed cross compiler hosted on the STAR access station, a CDC 6400 computer. The compiler was developed using a compiler writing system.

PL/STAR

A Structured Assembly Language

For the CDC STAR-100

CHAPTER 1

Introduction and Basic Concepts

1.1 Introduction to PL/STAR

PL/STAR is a structured assembly language (SAL) for the Control Data STAR-100 computer. PL/STAR draws on high level languages for the form of control structures and data structures, while providing the same access to machine features as is found in assembly language programming.

The purpose of PL/STAR is to provide systems programmers and other users a means to write assembler level code in a readable, maintainable and less error prone form.

One feature of code written in assembly language is its lack of an easily discernible organization. That is, line after line can appear on an assembly listing with no indication of the underlying program structure.

One goal of PL/STAR is to provide a means of writing assembly level code in a manner that makes the program structure clear.

Another goal of the language is to provide the user the advantages of high level language data structures. For this PL/STAR provides type declarations and a user defined type capability.

The first structured assembly language was PL360, implemented by Niklaus Wirth in 1965[3]. Since then there have been several such languages implemented. In addition to PL360 two other structured assembly languages, MIDAS[4] and PL85[5] have served as sources for PL/STAR.

While the languages vary in content, certain similarities do appear to be characteristic of structured assembly languages. All languages studied contain high level control structures, although the specific ones realized vary. All provide for variable declarations. While the declarations generally determine the amount of storage allocated for the variable, PL360 makes an additional distinction similar to the high level language concept of scalar type. Input/output are not treated in structured assembly languages.

Structured assembly languages are machine dependent.

1.2 The STAR-100 Computer

The Control Data STAR-100 is one of the largest and fastest scientific computers now available. Among the advanced features employed on the STAR are stream processing, virtual addressing and hardware macro instructions.

The STAR-100 provides an extensive vector handling capability. On the STAR a vector is a set of operands located in consecutive memory locations. Instructions are provided to perform a wide range of operations on vectors, from basic arithmetic to inner products and square roots[1], [2].

The word size on the STAR is 64 bits. Addresses are 48 bits long, and there are 256 general purpose registers. The STAR is a three address machine, so that the result field is specified explicitly and may differ from either operand.

The principal languages used on the STAR are FORTRAN, the assembly language META and SL/1. FORTRAN and SL/1 are oriented toward scientific programming. The bulk of system programming is done in assembly language or a FORTRAN like implementation language, IMPL.

1.3 Basic Concepts and Notation

This section defines elementary forms of the PL/STAR language.

1.3.1 Free Format Input

PL/STAR is a free format language. Statements may begin at any place on a card and extend over any number of cards. The amount of space between elements of a statement is determined by the user. The PL/STAR programmer, therefore, possesses the flexibility to produce visually coherent programs.

1.3.2 Identifiers and Reserved Words

An identifier is a symbol used to name a piece of data, a label or a constant. An identifier is a letter followed by a string, possibly empty, of letters, digits and underscores. While identifiers can be of arbitrary length, only the first ten characters are used by the compiler. For two identifiers to be distinct, then, they must differ within their first ten characters.

Examples:
first
max_size

Certain identifiers set forth in the language definition have an inalterable meaning and may not be redefined by PL/STAR users. These identifiers are called reserved words.

PL/STAR Reserved Words

AND	ARRAY	BIT
CALL	CHARACTER	CODE
CONST	DESCRIPTOR	DO
DREG	ELSE	ENDI
ENDM	ENDR	ENDW
ENTRY	GOTO	IF
INTEGER	LENGTH	MODULE
OF	OR	REAL
RECORD	REG	REPEAT
SDREG	SREG	THEN
TYPE	UNTIL	VALUE
VAR	WHILE	XOR

PL/STAR also contains one predeclared identifier. `BASE_REG` is a register that is initialized to hold the relocation amount for the module.

1.3.3 Data and Types

The quantities that PL/STAR units manipulate are its data. Each piece of data has three attributes that define it, its type, its storage class and its value.

PL/STAR provides two classes of types, scalar and structured. The scalar types are real, integer, bit, descriptor and character.

There are two structured types in PL/STAR, the array

and the record.

An array consists of some number of elements each of the same type. Arrays in PL/STAR are one dimensional. Elements of an array are referenced using an integer index.

A record is a grouping of elements that are not necessarily of homogeneous type. Each element of a record is termed a field and is named by an identifier. The elements of the record are referenced by the name of the record and the field identifier. No indexing is used.

Comments can be placed throughout the PL/STAR module to improve its readability. Comments may appear only between PL/STAR identifiers, numbers and symbols. The form of a comment is (* <any text string not including *> *).

Example:

```
(* A comment may take as much space as is
    necessary to make its meaning clear. *)
```

1.3.4 Grammar Specification

The syntax of the language is specified formally with a Backus-Naur form (BNF) grammar. The PL/STAR grammar is a set of productions that describe the permissible forms of a PL/STAR module.

The following sections employ a modified form of BNF to describe the syntax of the constructs under consideration.

The modifications consist of the addition of the following meta-symbols:

{ } grouping

[+] one or more repetitions

[*] zero or more repetitions.

CHAPTER 2

Language Description

2.1 Overview of PL/STAR Structure

This chapter provides an introduction to the design of PL/STAR.

The syntax of PL/STAR, like other SAL's, is similar to that of high level languages. For PL/STAR, SL/1[6] and PASCAL[7] served as sources, although the form of PL/STAR is adapted to suit the particular needs of its environment.

The compilable unit of PL/STAR is a module. From the user's perspective, a PL/STAR module is a separately compiled subroutine. At present there is no way for a job to begin execution within PL/STAR. This is a result of the view that PL/STAR is an addition to the present tools of systems implementors.

A PL/STAR module is divided into two basic sections. The first section, the declaration section, defines the data used by the module. The second section, the code section, defines the actions to be performed on that data.

2.1.1 The Declaration Section

PL/STAR provides three distinct types of data definition, constant declarations, variable declarations and user defined type declarations.

A constant declaration assigns to an identifier a value that may not be changed within the module. The type of the constant identifier is the same as that of the constant value to which it is bound. Neither a storage location nor a register is allocated for a constant declaration.

PL/STAR requires that all variables used in a program be declared. The absence of declaration by default aids responsible programming. At compile time, a declaration binds type and storage attributes to the variable identifier.

The declaration of type with the attendant type checking is a major departure from the assembly code level. Type checking is employed to promote reliable software. The compiler checks that operands of an expression are of compatible type. When an error in type is encountered the compiler signals an error.

There exists also a means to initialize variables if the user desires.

The final type of declaration is that of the user defined type. PL/STAR provides the user a limited facility to create his own data types.

2.1.2 The Code Section

The code section is a sequence of statements that defines the manner in which the data of the module is to be manipulated.

A salient feature of PL/STAR is the manner in which its statements resemble those found in higher level languages. The similarity is intentional. A high level syntax maps the linear structure of assembly language code into a form in which the flow of control is more apparent.

PL/STAR supplies three control structures, the IF, WHILE and REPEAT statements. Each statement has a unique terminator- ENDI, ENDW and UNTIL respectively. One result of the use of the terminators is the elimination of the dangling else problem in the IF statement.

The STAR-100 provides an instruction set of sufficient richness that an unacceptably complicated syntax would be the result of providing an operator for each machine instruction. To allow access to the full capability of the

machine, PL/STAR permits the user to incorporate symbolic assembly code into his module.

CHAPTER 3

Syntax

This section provides a detailed description of the parts of a PL/STAR module. BNF notation is used to provide a more formal description.

A PL/STAR module decomposes into six parts. In order of appearance the parts are:

- module header,
- constant declaration section,
- type declaration section,
- variable declaration section,
- initialization section,
- code section.

3.1 The Declaration Section

The symbol table is built within the declaration section. A description of the syntax used in the declaration sections follow.

3.1.1 Module Header

In BNF the module header is defined to be:

```
<module header> ::= MODULE <identifier> ;.
```

The identifier names the module. Its use, though, is restricted to the production of the loader tables. It is not defined to be an entry point, although the user may define an entry point with the same name.

Example:

```
MODULE plsort;
```

3.1.2 Constant Declarations

```
<constant declaration section> ::=
CONST <constant declaration>[+] | <epsilon>
```

```
<constant declaration> ::= <identifier> = <constant> ;.
```

The effect of a constant declaration is to add the identifier to the symbol table and associate with it the value and type of the constant.

The constant may be of any PL/STAR provided scalar

type.

Example:

```
CONST
    wordsize = 64;
    one = 1;
```

Both identifiers, `wordsize` and `one`, are entered into the symbol table with type integer and their respective values.

3.1.3 Type Declarations

```
<type declaration section> ::=
TYPE <type declaration>[+] | <epsilon>.
```

```
<type declaration> ::=
<type identifier> = RECORD <field list> ENDR;.
```

```
<field list> ::=
{<identifier> {, <identifier>}[*] : <scalar type> ;
}[+].
```

In PL/STAR the user may create his own data types. Currently, this facility is limited to record structures, although a general user definition capability is envisioned.

The record structure permits the description of complex quantities that can not be adequately represented with a scalar type.

Examples:

```

TYPE
    position = RECORD
    x_coord,
    y_coord : INTEGER;
    ENDR;

```

In this example position is declared to be a type with two integer fields. With this declaration, variables can be declared to be of type position.

```

weather = RECORD
    temperature : INTEGER;
    pressure : REAL;
    humidity : INTEGER;
    ENDR;

```

This declaration demonstrates the use of non-homogeneous type fields in a record. Weather is a type with three fields. Two fields are integer and the third is real.

3.1.4 Variable Declarations

```

<variable definition section> ::=
VAR <variable definition> [+].

```

```

<variable definition> ::=
<identifier> {, <identifier>}[*] : <type> STORAGE; .

```

All variables used in a PL/STAR module must be declared. The definition links a type and a residence to the variable identifier.

The type of a variable can be any predefined or user declared type.

A PL/STAR variable may have a residence in a register

or in memory.

All PL/STAR structured variables must be memory resident. Elements of array and records are allocated consecutive memory locations. The declared bound of an array denotes the number of elements in the array and the first element of each array has index zero.

The only operations that the language directly provides for use with memory resident variables are loading and storing.

The declaration of a variable to be register resident permits the user to specify that the register variable be saved and restored by called procedures or initialized.

These actions are specified by the reserved word used to declare a variable register resident:

SREG - register is to be saved and restored by called procedures

DREG - register is to be initialized

SDREG - register is to be saved, restored and initialized

REG - register is not saved, restored or initialized.

The notation and conventions for the above functions are based upon the set of STAR Standard Procedures [8].

Examples:

```

VAR
    max, min : INTEGER REG;
    prime    : REAL SDREG;
    p1, p2   : position;
    mem      : INTEGER;
    a        : ARRAY [10] OF INTEGER;
    rec      : RECORD
                fi, f2 : integer;
            ENDR;

```

The registers corresponding to max and min will contain integers. Neither register is saved or restored. prime is declared to be a register that will hold a real and will be saved and restored. p1 and p2 are of user declared type position. The elements of array a are a[0] through a[9]. rec demonstrates the use of a record declaration in the variable declaration part.

3.1.5 Initial Section

```

<initial section> ::=
VALUE {<identifier> = <constant>;}[*] | <epsilon>.

```

An initial directive causes loader directives to be generated that will initialize the variable to the desired value. Type checking is performed to enforce compatibility.

Example:

```

VALUE
    max = 100;

```

Before execution, the loader will initialize max to the given value.

3.2 The Code Section

The code section is a list of executable statements introduced by the reserved word CODE. Statements comprising a list are separated by semicolons. PL/STAR statements are the assignment, GOTO, WHILE, REPEAT, IF, CALL, ENTRY and empty statements.

3.2.1 Assignment Statement

<assignment statement> ::= <left hand side> := <expression>.

The effect of an assignment statement is the alteration of the value of a variable designated by the left hand side to the value of the expression on the right hand side of the assignment operator.

The left hand side may refer to either a memory or register resident variable.

A memory resident variable can be referenced either by its name or by indirect addressing, that is, in PL/STAR, by a register variable that contains the address of the memory resident variable followed by a caret (^).

A register resident variable is referenced by its name.

While the forms that an expression may take are varied, most of the forms reduce to one machine instruction.

Certain loads and stores require two. The limited nature of the PL/STAR expression contrasts sharply with that found in other PL360 derivative languages that generally permit expressions of arbitrary length.

Arbitrary length expressions were avoided for three reasons.

A significant feature of the design of PL/STAR is the deliberate omission of register management in favor of allowing the user to control the allocation of registers. A result of this policy is the absence of registers to hold intermediate results in expression evaluation. Expressions, therefore, had to be limited to a form that maps into a single machine instruction.

An unavoidable exception to the single instruction rule occurs with certain load and store sequences in which a register is needed to hold an address. In this case a register from a group of designated temporary registers is used to hold the address.

The second reason is that structured assembly languages in general do not include parentheses and treat all operators as having equal precedence. Thus statements are translated strictly left to right. It is quite possible, therefore, that the meaning of a multioperator expression to a structured assembly language translator differs from the meaning a programmer intended.

The third reason is pragmatic. It was felt that the

form employed in PL/STAR would be sufficiently powerful to satisfy most users.

3.2.1.1 Conditions

A condition is a relationship between two register resident variables. The relational operators are >, >=, <, <=, = and <>. Each condition has a boolean value. The value of the condition is true if the stated relationship holds between the variables, false if the relationship does not exist.

3.2.1.2 Expressions

An expression takes one of four forms. A constant generates the appropriate load instruction. A memory resident variable generates loads. A register resident variable generates store or register transfer instructions.

An expression can also be of the form <operand1> operator <operand2>. Both operands must be register resident. The operator can be arithmetic (+, -, *, /) or boolean (and, or, xor) or relational (<, <=, >, >=, =, <>).

Examples:

```
pl.x_coord := max;  
loc := loc + increment;  
loc^ := min;
```

The first statement shows an assignment into a field of a record. The value of max is stored in the memory location assigned to field x_coord of pl. The second example adds the value of increment to the value of location. The third statement places the value of min into the storage location pointed to by loc.

3.2.2 If Statement

The IF statement is available in both the IF-THEN and IF-THEN-ELSE forms. The BNF form of an IF statement is:

```
<if statement> ::= IF <condition> THEN <statement list>
ENDI |
IF <condition> THEN <statement list> ELSE <statement
list> ENDI.
```

Examples:

```

IF cand > max THEN
  max := cand;
ENDI

IF loc < stop_place THEN
  loc := loc + increment;
  cand := loc^;
ELSE
  answer := max;
ENDI

```

The examples illustrate the two versions of an IF statement. The semicolon before the ELSE is legal in PL/STAR. An empty statement following the semicolon completes the parse of the statement list.

The effect of the IF statement is the same as that in high level languages.

3.2.3 Iterative Statements

There are two iterative statements in PL/STAR, the WHILE and the REPEAT statements.

```

<while statement> ::= WHILE <condition> DO <statement
list> ENDW.

```

```

<repeat statement> ::= REPEAT <statement list> UNTIL
<condition>.

```

Examples:

```

WHILE loc < stop_place DO
  loc := loc + increment;
ENDW

```

```

REPEAT
  loc := loc + increment;
UNTIL loc >= stop_place

```

These statements operate in the same manner as they do in other high level languages.

3.2.4 GOTO Statement

<goto statement> ::= GOTO <label>.

Example:

```
GOTO maxfound
```

A GOTO statement transfers control to the statement labelled by <label>. A label is an identifier, and a statement is labelled by preceding the statement with the label and a colon. Labels are not declared.

Since PL/STAR is a one pass compiler, the relative address corresponding to the label may not be known when a GOTO is encountered.

This situation can cause less than optimal code to be generated. The STAR provides two types of branch instructions, one type 64 bits long, the other 32 bits long. Branching is expressed as an offset from the current contents of the program counter. For space and time considerations, the 32 bit branch instruction is preferred;

however, it cannot express as large a jump as the full word instruction. When a label with an undetermined address is encountered in a GOTO the full word branch is generated although the half word instruction may have sufficed.

GOTO's are permitted to branch only within the module. An attempt to branch to a point outside the module results in a compile time error. It is possible to branch into the bodies of control structures, although the practice is discouraged.

3.2.5 Empty Statement

Empty statements contain no symbols and cause no action. They can be labelled and be the object of branches, however. Empty statements occur more frequently than one might suspect. Consider the following WHILE statement in which C is a condition and S1 and S2 are statements:

```
WHILE C DO
    S1;
    S2;
ENDW
```

There is an empty statement between S2; and ENDW. Since semicolons separate statements and ENDW is not part of the statement list, there must be another statement after S2. The only possible statement is the empty statement.

3.2.6 Call Statement

The CALL statement transfers control to an external procedure and passes it parameters. The form of a CALL statement is:

```
<call statement> ::= CALL <procedure name> [|
CALL <procedure name> ( parameter {, parameter}[*]).
```

A parameter can be a constant or a register variable, which passes the contents of the register variable, or a register variable in brackets, which passes the contents of the address pointed to by the register variable.

Examples:

```
CALL extproc(max, [loc], 5)
CALL proc2
```

To execute the first CALL statement, control is transferred to external entry point extproc. The values pointed to by parameter list entries are the value of max, the value contained in the memory location whose address is in loc and the value 5.

The second statement transfers control to a point which takes no parameters.

3.2.7 Entry Statement

An ENTRY statement, which must be labelled, establishes an entry point in the PL/STAR module. The entry point is

referenced by the label of the ENTRY statement. The form of an ENTRY statement is:

```
<entry statement> ::=
```

```
ENTRY | ENTRY (parameter {, parameter}[*]).
```

The ENTRY statement invokes the standard procedure entry sequence. The caller's registers are saved and the called module's registers are loaded.

A parameter is either a register variable or is empty. The register parameters are loaded with the values of the corresponding arguments from the calling procedure.

Example:

```
sort_proc:  
    ENTRY (max, min)
```

Max and min are loaded with the values contained in the caller's parameter list.

CHAPTER 4

Implementation of the PL/STAR Compiler

This chapter supplies a description of the structure and implementation of the PL/STAR compiler.

PL/STAR is a one pass syntax directed compiler written in PASCAL. The compiler is built using a compiler writing system developed by Professor Michael Donegan of William and Mary. A discussion of the system is in chapter six.

The remainder of the chapter is divided into three sections. The first is a description of the symbol table. Following that is a discussion of the compiler functions. The chapter concludes with an overview of the form of the loader tables generated for the STAR.

4.1 The Symbol Table

The most conspicuous data structure in the PL/STAR compiler is the symbol table, in which information describing the data of the PL/STAR module is stored.

The symbol table for PL/STAR is a straightforward mechanism, a list of entries corresponding to the declared data. Entries are generated for variables and constants, and new entries are added to the front of the list.

Information kept on each symbol is its name, type and storage class. For arrays the number of elements is saved; for records a pointer to the record definition is kept. The register allocated is recorded for register resident variables.

Symbol Table Entry Fields

IDFIELD	name of symbol
KIND	type
STORE	storage residence
ARRAYSIZE	bound of arrays
ADDRESS	relative bit address in data base
REGISTER	register assigned to register variables
INITIALED	true if variable is initialed
NEXT_SYM	pointer to next symbol in table
	a case variant that can hold a value for a scalar type or a pointer to a record type

A new entry in the symbol table is manufactured in procedure ENTER. ENTER first determines that the symbol has

not been previously defined and then calls procedure CREATE, which generates a new symbol table entry and fills its fields with spurious values. ENTER then fills in the proper values and fixes the pointers.

Separate lists maintain information on labels and entry and external entry points. An entry in the list of labels consists of the label identifier, the relative address and a pointer to a list of addresses at which the label is used in forward branches.

Information kept on entry points defined within the PL/STAR module is the identifier of the entry point and its relative address. For external entry points all the addresses from which it is called are also saved.

4.2 Syntax Analysis and Code Generation

The PL/STAR compiler is in two parts. In the first part the symbol table is built, the syntax analyzed and code generated. In the second half the loader tables that are communicated to the STAR are generated. This section deals with the first half.

4.2.1 Parsing Routines

The compiler generating system supplies two procedures that execute the parse. One, READTABS, initializes the parser tables.

The other, procedure PARSE, is the driver of the parse. It maintains the stack of symbols, causes tokens to be read, and directs the execution of semantic actions.

The form of PARSE is a loop. Upon getting the next input symbol, or token, PARSE determines the action corresponding to that symbol and the current state. If the action requires a shift, the symbol is pushed onto the stack and a flag is set to get the next symbol.

If reductions are necessary, the appropriate semantic actions are invoked and the requisite number of states are popped from the stack. Each reduction results in a new action.

If necessary, the next symbol is scanned.

This process continues until either an accept state or an error state from which recovery is impossible is entered. An accept state is entered after the input program has been parsed successfully.

4.3 Semantic Actions

The second procedure called by PARSE is SYNTHESIZE. SYNTHESIZE controls all the semantic actions and, therefore, the code generation properties of the compiler.

SYNTHESIZE is a large case statement. The case labels correspond to the rules of the grammar. It is important to remember that in a syntax directed compiler semantic actions can be invoked only at the end of a rule.

More than one rule is normally required to describe a PL/STAR form. What follows is a discussion of the actions taken by the compiler for the various statements of PL/STAR.

4.3.1 Module Header

The sole action associated with the module header is the saving of the module name for use in generating loader tables.

4.3.2 Constant Declarations

For each constant declaration a symbol table entry is generated. In addition to the notation that the entry describes a constant, the entry includes the identifier and the type and value of the constant.

4.3.3 Type Definitions

In the type section a list, headed by `FIRST_TYPE`, is built. Each entry in the list corresponds to a user defined type. A list entry consists of the type identifier, a pointer to the type definition and a pointer to the next list entry.

The definition of a record is itself a list. An entry in a record definition list contains the field identifier and its type. Each element of a record definition may have as many as ten identifiers declared to be a single type. To handle this situation, the compiler maintains a list of the identifiers, and an entry for each identifier in that list is generated when the type is known.

4.3.4 Variable Declarations

Storage and register allocation is accomplished in the variable declaration section.

To facilitate the saving of registers, all `SREG` and `SDREG` type variables must occupy consecutive registers and words in the data base. To do this `SREG` and `SDREG` variables are allocated registers beginning with register 20 (hex), the first general purpose register available to the user, and word 0 of the data base. As the declarations are

encountered, the compiler assigns offsets from the first of the type for REG and DREG type registers. An offset in bits is kept for memory resident quantities also. After the last declaration is processed, the proper amounts are added so that the REG and DREG variables follow the SREG and SDREG variables and the memory resident variables follow the DREG and REG ones.

4.3.4.1 Arrays and Records

For an array the proper number of memory words are allocated. The address contained in the symbol table for the array is the address of the first element of the array.

A list of identifiers may be declared to be of a record type enumerated in the variable declaration section. The same list as that built for a record in a type declaration is constructed, and that list is walked through to allocate memory for the fields.

When a user defined type is encountered, the compiler finds its entry in the list of user defined types and its definition is traversed to allocate storage for the fields.

4.3.5 Initial Section

In a value directive the symbol table is searched for the identifier, a flag is set to show the loader table generation routines that an initialization is to be done and the initial value saved.

4.3.6 Code Section

At this point in the parse the symbol table has been completed. The compiler uses the information contained in the symbol table to generate machine instructions in the code section.

A discussion of the actions corresponding to PL/STAR forms follow.

4.3.6.1 Labels

When a label is encountered, the compiler searches the list of labels for an entry that matches the label identifier. If a match is not found, the label definition has preceded its use in any GOTO. The compiler adds a new entry in the label list containing the label identifier and the relative address of the label.

If a match does occur, the label was used in a forward

branch. The compiler can then enter the proper relative address in the list entry.

4.3.6.2 GOTO Statement

A GOTO statement maps into one of the two types of unconditional branch instruction that the STAR supplies. Whether the halfword or fullword instruction is used depends upon the distance of the branch destination from the branch instruction.

The compiler searches the label list for the label. If it has been defined, the halfword branch instruction is generated if the branch destination is no than 255 halfwords distant. If the destination is more than 255 halfwords away, the fullword branch instruction is generated.

If the label has not been defined, an entry in the label list is generated for the first occurrence of a label. An entry in the list of forward references is made so that after parsing the code generated can be altered to show the proper branch displacement. A fullword branch instruction must be generated since the final displacement is not known.

Example:

Assume label1 is determined to be address n and the value of the instruction counter is i.

The translation of the statement GOTO label1 is

BADF	i-n	if n = i and i-n < 256
IBXEQ,F	n-i,0,0	if n > i
IBXEQ,B	i-n,0,0	if n < i and i-n >= 256

The format used in this paper to show branch instructions differs from standard META. Following the instruction mnemonic is the branch displacement. For conditional branch instructions the registers to be compared follow the displacement.

BADF is the halfword branch instruction generated when the location of the branch is less than 256 halfwords from the value of the instruction counter. In PL/STAR the BADF is always an unconditional branch instruction.

The IBX series are fullword conditional branch instructions. Following the IBX is a two letter mnemonic identifying the condition being tested. The ,F indicates a forward branch and the ,B a backward branch. The IBX instruction causes the two registers to be compared and the branch taken if the tested condition is met.

The two IBX instructions above are in effect unconditional branches. Register 0, the contents of which is always zero, is tested for equality with itself.

4.3.6.3 Assignment Statement

The code generated by the compiler for an assignment statement depends upon the form of the statement.

1. <register identifier> := <register identifier>
<operator> <register identifier>.

This form translates into a halfword instruction in which the three registers are the operands of the instruction. The specific instruction generated is determined by the operator and the types of the registers.

Example:

r1 := r2 operator r3 generates the halfword instruction OPCODE r2,r3,r1, in which OPCODE is determined by the following table:

OP/TYPE	INTEGER	REAL	BIT
+	63	62	
-	67	66	
*	3D	6B	
/		6F	
AND			2D
OR			2E
XOR			2C

2. <register identifier> := <operand>.

If the operand is a constant, a load immediate is generated. If it is another register, a register to register assignment is performed. If the operand is a storage resident variable, a load instruction is generated.

A storage resident quantity can be specified in two ways. A register identifier followed by a caret, indicating indirect addressing, causes one instruction to be generated.

A variable declared to be storage resident, however, requires an additional instruction to load a temporary register with the address of the operand.

Examples:

<code>r1 := 6</code>	generates	<code>ES r1,6</code>
<code>r1 :=65536</code>	generates	<code>EX r1,65536</code>
<code>r1 := r2</code>	generates	<code>RTOR r2,0,r1</code>
<code>r1 :=mem</code>	generates	<code>ES TEMPREG,address(mem)</code> <code>LOD TEMPREG,BASEREG,r1</code>
<code>r1 :=r2^</code>	generates	<code>LOD r2,0,r1</code>
<code>r1 :=r2^[r3]</code>	generates	<code>LOD r2,r3,r1</code>

The ES and EX instructions both place the second operand into the register designated by the first operand. ES is a halfword instruction used for integers less than 65536; EX is a fullword instruction is used for larger integers. RTOR is a register transfer instruction.

3. `<register identifier> := LENGTH(<register identifier>)`.

An instruction that replaces the contents of the result register with the leftmost 16 bits, called the length field, of the operand register is generated.

Example:

<code>r1 := LENGTH(r2)</code>	generates	<code>LTOR r2,0,r1</code>
-------------------------------	-----------	---------------------------

4. `<memory resident variable> := <operand>`.

For this form store instructions are generated. If the result field is specified by indirect addressing, only one instruction is needed. If the result field is a memory resident variable, however, the extra instruction to load a temporary register with its address is needed.

Examples:

```

mem := r2      generates  ES TEMPREG,address(mem)
                  STO r2,BASEREG,TEMPREG

r1^ := r2      generates  STO r2,0,r1

```

4.3.6.4 Conditions

One part of each control structure is the controlling condition, a relationship between two registers. The compiler maintains pointers to the two symbol table entries and a record of the relational operator.

See section 3.2.1.1.

Examples:

```

cand < max     generates  IBXGE cand,max

loc >= stop    generates  IBXLT loc,stop

```

4.3.6.5 While Statement

The first action specific to a WHILE statement occurs

after the reserved word DO is recognized. The relation of the condition is complemented, that is, equal becomes not equal, greater than becomes less than or equal etc. A fullword branch using the complemented relational and the two registers is generated so that a branch around the WHILE statement occurs if the original condition is false. The address of the branch is pushed onto a stack of addresses.

The statement list of the WHILE, which may itself contain other control structures, is compiled next.

When the ENDW is seen, the address of the branch instruction at the head of the statement is popped from the stack. An unconditional branch to there is generated as well as the information needed to fix the displacement of the WHILE branch.

Example:

	halfword address	
WHILE r1=r2 DO	i,i+1	IBXNE,F n-i+1,r1,r2
<statement list>	i+2 n-1	code for statement list
ENDW	n	BADF n-i

An IBX instruction is used in place of the BADF if $n-i > 255$. The body of the WHILE statement is executed only as long as the condition is true. An IBX instruction is generated, therefore, that will cause a branch about the WHILE body when the condition is false. The condition of

equality in the PL/STAR code thus becomes the IBXNE of the generated code. After the code for the statement list of the WHILE has been completed, an unconditional branch back to the condition test is made.

4.3.6.6 Repeat Statement

The treatment of the REPEAT statement is similar to that of the WHILE statement. The current relative address is pushed onto the stack when the word REPEAT is seen, and the statement list compiled.

After the concluding condition has been done, the stack of addresses is popped to obtain the address of the first statement in the statement list. The condition's relation is complemented and the branch generated to return to the top of the REPEAT.

Example:

	halfword	
	address	
REPEAT		
<statement	i	code for
list>	n-1	statement list
UNTIL r1=r2	n,n+1	IBXNE n-i,r1,r2

The expansion of a REPEAT statement consists of the generation of the code for the statement list followed by a conditional branch to the first instruction of the REPEAT statement list if the condition is false.

Since the termination condition for the example is the equality of the two registers, the branch tests for inequality.

4.3.6.7 If Statement

The IF statement is more complicated since it may or may not contain an ELSE part. After the THEN, the current relative address is stacked, the relational complemented and a branch instruction generated. The statement list to be executed if the condition is true is compiled.

After the statement list is completed, the next symbol can be either an ELSE or an ENDI. In either case the address of the IF branch is popped and the information to fix its displacement generated.

If the symbol is an ENDI, the statement is completed.

If the symbol is an ELSE, the current address is stacked and an unconditional branch about the ELSE code is generated. The ELSE statement list is compiled. The address of the branch about the ELSE is popped and the information to fix its displacement is generated.

Examples:

	halfword address	
IF r1=r2 THEN	i,i+1	IBXNE,F n-i+1,r1,r2
<statement list>	i+2 n	code for statement list
ENDI		

The IBXNE,F statement causes a branch to the statement following the body of the IF statement if the if condition is not met.

IF r1=r2 THEN	i,i+1	IBXNE,F j+2-i,r1,r2
<statement list>	i+2 j-1	code for statement list
ELSE	j,j+1	IBXEQ,F n-j+1,0,0
<statement list>	j+2 n	code for statement list
ENDI		

In this case the IBX condition will branch to the first instruction of the ELSE part if the condition is not met, i.e. $r1 \neq r2$. After the THEN code, an unconditional branch is generated to skip the ELSE part.

4.3.6.8 Call Statement

For a CALL statement the compiler adds the external entry point to the list of entry points if it does not already appear there, and adds the current address to the list of addresses from which the entry point is called.

The next step is to process the parameters. Parameter addresses are placed in consecutive locations in memory. A descriptor placed in a register allows the called procedure to reference them.

There are three forms of parameters. If a parameter is a constant, the address of a memory location containing the constant is placed into the parameter list.

The second type of parameter is a register identifier. The contents of the register is placed into its location in the data base and the address of that location is placed in the parameter list.

The third form is a register identifier in brackets. In this case the register contains the address of the parameter. A store instruction is generated to place the address contained in the register into the parameter address list. Finally the code for a procedure call is generated. An instruction is generated that will place the address of the entry point into register 1E (hex). This address is not known at compile time; the loader places the proper address into the instruction. The parameter descriptor is placed

into register hex 17. A branch and link instruction sets the return register and branches to the called procedure.

Code for Procedure Call

```
RTOR entrypoint+1,0,1E    load link register
RTOR parameterdesc,0,17  load parameter descriptor
BSAVE 1A,0,entrypoint    branch and set return
```

4.3.6.9 Entry Statement

In an ENTRY statement the associated label is added to the list of entry points, and the code for an entry is generated. The sequence of instructions saves the caller's registers and designates which of the called module's are to be saved in the case of a call. Finally the registers that are to be initialized are loaded with their initial values.

The next step is the handling of parameters. All the formal parameters must be registers. Into these registers are placed the contents of the parameter list entry for the corresponding actual parameter. Since this entry will normally be an address, the load instruction to fetch the value of the actual parameter is the responsibility of the user.

Code to Establish an Entry Point

```
SWAP 0,15,1C  save caller registers
ELEN 1F,0
RTOR 1C,0,1D  update stack pointers
RTOR 1B,0,1C
ELEN 1C,x      x is number of registers to be saved
IS  1B,x*64   Increment the dynamic stack pointer
ELEN 1E,z      z is number of registers to be loaded
                for execution
SWAP 1E,14,0  Load registers
```

4.3.6.10 Completion of Code Generation

Upon encountering the reserved word ENDM the standard procedure exit sequence is generated.

Exit Sequence

```
SWAP 1D,15,0  restore caller's registers
LSDFR 060000  load data flag register
BADF 1A      return control to caller
```

After parsing has been completed, the forward branches associated with WHILE, IF and GOTO statements must have the proper displacement added before the loader tables can be generated.

At the conclusion of the WHILE and IF statements and after the labels of forward branching GOTO's have been defined, entries are made in a list, ordered by increasing relative address, of displacements that have to be added to the instructions contained in those addresses. It is then a simple matter to transfer code, as halfwords, from the file to which the code was generated originally to a new file with the proper displacements added.

Following this operation the listing and symbol table are printed.

4.4 Generation of Loader Tables

PL/STAR generates four tables to create a load module. An overview of their contents follows.

The MODULE table is the header for the load module. It contains the name of the module, the length of the tables, the length of the code, the length of the database and pointers to the other modules.

The CODE table contains the relocatable code.

The EXT ENTR table contains a list of entry point and external entry point names.

Following the list of names is a list of one word descriptors for the names in the list. A descriptor contains two fields. The leftmost 16 bits describe the type of the name, entry or external entry. The remaining 48 bits contain the relative bit address in the code for entry points and 0 for external names.

The interpretive data initialization table, named INT DATA in the tables, contains directives to initialize storage locations[10].

CHAPTER 5

Module Listings and Sample Modules

This chapter describes the format of a PL/STAR listing and presents two sample modules that have been executed on the STAR computer.

5.1 The Listing Format

PL/STAR supplies the user a listing of the code generated for his module. This information appears to the left of the PL/STAR source line for which the code is generated.

The first field on a line is the relative bit address of the instructions generated for the source line. Following the relative bit address is one or two halfwords of code. These fields are printed only if instructions have been generated for that source line. If more than two

halfword instructions have been generated for a line, the remaining instructions are printed on succeeding lines before additional source.

Following the generated instructions are the number of the source line and the PL/STAR source line itself.

After the source has been listed a symbol table directory is printed. The type, storage residence and database address of each variable is printed. For each register resident variable, the register assigned to the variable is printed. For array variables the size of the array is printed.

To conclude the symbol table a list of all labels and their relative addresses is printed.

5.2 The Sample Modules

The two examples provided are both called from SL/1 routines, listings of which are also provided. The SL/1 modules initialize the data on which the PL/STAR modules operate, call the PL/STAR modules and print results.

The first PL/STAR module, PLSORT, sorts into descending order an integer vector of arbitrary length.

No code is generated in the declaration sections as the symbol table is being built. At line 24 the first eight halfword instructions comprise the standard entry sequence.

To load the parameters the address of the parameter list is placed into a temporary register. The contents of the parameter list are then placed into the corresponding registers with individual load instructions. Succeeding parameter list values are accessed by executing an add immediate instruction on the temporary register.

Note that the four instructions to set the variable LIM could have been expressed in one instruction if PL/STAR did not limit expression length.

The remainder of the module is just a loop with another loop nested within it. The outer loop is delimited by the branch instructions on lines 34 and 63. These instructions were generated during the parsing of the WHILE statement. The first segment of the loop initializes the variables MAX, CTR and SWITCH using a load, an add and a register to register assignment respectively.

The second segment of the loop is the inner loop defined by the branches on lines 42 and 53. The loop compares each remaining member of the vector to the value of MAX. If an element greater than MAX is found, that element becomes the new value of MAX and SWITCH is assigned its location.

The final portion of the outer loop, on lines 55 through 63 accomplishes the exchange and increments the index.

```

1  -MODULE PLTEST BEGIN PLTEST;
2
3  -EXTERNAL PROCEDURE (INTEGER VECTOR(10), INTEGER) PLSORT;
4
5  -INTEGER VECTOR I(10) IV;
6
7  -INITIAL (IV = \7, 3, 1, 5, 9, 6, 12, 14, 0, 8\);
8
9  -PROCEDURE PLTEST;
10
11  -WRITE # '//////////', ' BEFORE SORTING, VECTOR IS '#;
12  -WRITE (IV) # 3X, 15 #;
13  -CALL PLSORT(IV,10);
14  -WRITE # /, ' AFTER SORTING, VECTOR IS '#;
15  -WRITE (IV) # 3X, 15 #;
16
17  -ENDP;
18  -ENDM;
19
20

```

AMOUNT OF CODE GENERATED IS 36 (DECIMAL) WORDS.

DATA BASE SIZE IS 109 (DECIMAL) WORDS.

COMPILATION TIME : 0.38 SECOND(S)

MODULE PLSORT;

(* THIS MODULE SORTS AN INTEGER VECTOR INTO
DESCENDING ORDER. THE ARGUMENTS PASSED TO THE
ARE THE ADDRESS OF THE FIRST ELEMENT OF THE
VECTOR AND THE SIZE OF THE VECTOR. *)

VAR

SIXTYFOUR : INTEGER REG;

FIRST, NUM,

LIM, SWITCH,

IX : INTEGER REG;

MAX, CTR,

ONE, CAND,

TEMP : INTEGER REG;

VALUE

ONE = 1;

CODE

PLSORT: ENTRY(FIRST, NUM);

000000 7000151C 2A1F0000

000040 781C0010 7818001C

000080 2A1C0007 3F1B01C0

0000C0 2A1E000C 701E1400

000100 78170003 7E030022

000140 3F030040 7E030023

000180 3F030040

0001A0 3E210040

0001C0 78220026

000150 7E23G024

SIXTYFOUR := 64; (* BITS PER WORD *)

IX := FIRST; (* IX HAS ADDRESS OF FIRST

VECTOR ELEMENT *)

(* SET LIM TO ADDRESS OF LAST VECTOR ELEMENT *)

LIM := NUM^;

```

0000200 67242924
0000220 30242124
0000240 63222424
0000260 82040026 00132400
00002A0 78260025
00002C0 7E260027
00002E0 63262128
0000300 85040028 00092400
0000340 7E28002A
0000360 8404002A 00042700
00003A0 782A0027
00003C0 78280025
00003E0 63282128
0000400 33460008
0000420 7E26002B
0000440 7F25002B 7F260027
0000480 63262126
00004A0 33460012
00004C0 701D1500 38060000
0000500 3340001A
31 LIM := LIM - ONE;
32 LIM := LIM * SIXTYFOUR;
33 LIM := FIRST + LIM;
34
35 WHILE IX < LIM DO (* NO NEED TO COMPARE LAST ELEMENT
36 WITH ITSELF *)
37 SWITCH := IX; (* SWITCH IS ADDRESS OF LOCATION OF
38 LARGEST VALUE FOUND IN VECTOR *)
39 MAX := IX^;
40 CTR := IX + SIXTYFOUR;
41
42 (* TEST REMAINING VALUES IN VECTOR *)
43 WHILE CTR <= LIM DO
44 CAND := CTR^;
45
46 IF CAND > MAX THEN
47 (* SET NEW MAX AND NOTE ITS LOCATION *)
48 MAX := CAND;
49 SWITCH := CTR;
50 ENDI;
51
52 (*SET CTR TO ADDRESS OF NEXT VECTOR ELEMENT*)
53 CTR := CTR + SIXTYFOUR;
54 ENDW;
55
56 (* EXCHANGE MAX AND IX^ *)
57 TEMP := IX^;
58 SWITCH^ := TEMP;
59 IX^ := MAX;
60
61 (* SET IX TO NEXT ELEMENT *)
62 IX := IX + SIXTYFOUR;
63 ENDW;
64
65 ENDM;

```

SYMBOL TABLE

SYMBOL	REGISTER	ADDRESS	ARRAY SIZE	TYPE	STORAGE
BASE_REG	00020		0	INTEGER	SDREG
TEMP	0002B	704		INTEGER	REG
CAND	0002A	640		INTEGER	REG
GNE	00029	576		INTEGER	REG
CTR	00028	512		INTEGER	REG
MAX	00027	448		INTEGER	REG
IX	00026	384		INTEGER	REG
SWITCH	00025	320		INTEGER	REG
LIM	00024	256		INTEGER	REG
NUM	00023	192		INTEGER	REG
FIRST	00022	128		INTEGER	REG
SIXTYFOUR	00021	64		INTEGER	REG

LABELS

PLSORT 0000000

BEFORE SORTING, VECTOR IS

- 7
- 3
- 1
- 5
- 9
- 6
- 12
- 14
- 0
- 8

AFTER SORTING, VECTOR IS

- 14
- 12
- 9
- 8
- 7
- 6
- 5
- 3
- 1
- 0

The purpose of the second module, TESTMOD, is to demonstrate the use of a record.

The entry code is handled in the same manner as in the first sample module. The assignments into the record fields are examples of cases in which a temporary register is needed for a store.

The assignment `SIZE := SIZE^` is necessary to place the actual size of the vector into the register since the parameter list contains the address of the parameter.

The content of the IF statement has been manipulated to show an IF-THEN-ELSE construct.

```

1  -MODULE PLTEST2 BEGIN PLTEST2;
2
3  -EXTERNAL PROCEDURE (INTEGER VECTOR(5), INTEGER, INTEGER,
4  -INTEGER) TESTMOD;
5
6  -INTEGER VECTOR (5) IV;
7
8  -INITIAL (IV = \3, 2, 1, 5, 4\);
9
10 -PROCEDURE PLTEST2;
11 -INTEGER MAX, PLACE;
12
13 -MAX := -12;
14 -PLACE := -12;
15
16 -CALL TESTMOD(IV, 5, MAX, PLACE);
17 -WRITE (MAX, PLACE) #/////////, MAXIMUM VALUE =, I3,
18   , IN LOCATION, I3 #;
19
20 -ENDP;
  -ENDR;

```

AMOUNT OF CODE GENERATED IS 24 (DECIMAL) WORDS.

DATA BASE SIZE IS 107 (DECIMAL) WORDS.

COMPILATION TIME : 0.38 SECOND(S)

```

1 MODULE TESTMOD;
2
3 (* THIS MODULE ILLUSTRATES THE USE OF A RECORD. THE
4 MODULE RETURNS THE VALUE AND LOCATION OF THE
5 LARGEST MEMBER OF AN INTEGER VECTOR. *)
6
7 TYPE
8
9   RECTYPE = RECORD
10     MAX, PLACE : INTEGER;
11   END;
12
13 VAR
14
15   MAX, MAX_SITE, TEMP : INTEGER REG;
16
17   WORDSIZE, SIZE, RUN, CTR : INTEGER REG;
18   CAND, ONE, IX1, NUM, PLACE : INTEGER REG;
19
20   REC : RECTYPE;
21
22 VALUE
23   WORDSIZE = 64; ONE = 1; CTR = 2;
24
25 CODE
26   TESTMOD: ENTRY(IX1, SIZE, MAX_SITE, PLACE);

```

```

27   RUN := IX1; (* SET RUN *)
28

```

```

000000 7D00151C 2A1F0000
000040 781C001D 781B001C
000080 2A1C0007 3F1B01C0
0000C0 2A1E000D 7D1E1400
000100 7817C003 7E03002A
000140 3F030040 7E030025
000180 3F030040 7E030022
0001C0 3F030040 7E03002C
000200 3F030040

```

```

000020 782A0026

```

```

29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
0000240 7E260021
0000260 3E030340 7F032029
0000280 3E030380 7F032021
00002F0 7E250025
0000300 63262426
0000320 85040027 001G2500
0000360 7E260028
0000380 84040028 000A2100
00003C0 78280021
00003E0 3E030340 7F032027
0000420 3E030380 7F032028
0000460 63272927
0000480 80040000 00030000
00004C0 63272927
00004E0 63262426
0000500 3346000F
0000520 3E030380 7E032023
0000560 7F220023
0000580 3E030340 7E032023
00005C0 7F2C0023
00005E0 701D1500 38060000
0000620 3340001A

(* SET MAX AND REC TO FIRST VECTOR ELEMENT *)
MAX := RUN^;
REC.PLACE := ONE; (* PASCAL-LIKE NOTATION *)
REC.MAX := MAX; (* FOR RECORDS *)

(* LOAD SIZE WITH NUMBER OF ELEMENTS IN VECTOR *)
SIZE := SIZE^;
RUN := RUN + WORDSIZE;

WHILE CTR <= SIZE DO
  CAND := RUN^;
  IF CAND > MAX THEN
    (* SET NEW MAX AND UPDATE RECORD *)
    MAX := CAND;
    REC.PLACE := CTR;
    REC.MAX := CAND;
  CTR := CTR + ONE;
ELSE
  CTR := CTR + ONE;
END;

END;

RUN := RUN + WORDSIZE;
ENDW;

(* RETURN MAX AND LUC VALUES FROM RECORD *)
TEMP := REC.MAX;
MAX_SITE^ := TEMP;
TEMP := REC.PLACE;
PLACE^ := TEMP;

ENDM;

```

SYMBOL TABLE

SYMBOL REGISTER ADDRESS SIZE TYPE STORAGE

SYMBOL	REGISTER	ADDRESS	SIZE	TYPE	STORAGE
BASE_REG	00020	0		INTEGER	SUREG
REC				RECORD	
MAX		896		INTEGER	
PLACE		832		INTEGER	
PLACE	0002C	768		INTEGER	REG
NUM	0002B	704		INTEGER	REG
IX1	0002A	640		INTEGER	REG
ONE	00029	576		INTEGER	REG
CAND	00028	512		INTEGER	REG
CTR	00027	448		INTEGER	REG
RUN	00026	384		INTEGER	REG
SIZE	00025	320		INTEGER	REG
WORDSIZE	00024	256		INTEGER	REG
TEMP	00023	192		INTEGER	REG
MAX_SITE	00022	128		INTEGER	REG
MAX	00021	64		INTEGER	REG

LABELS

TESTMOD 0000000

MAXIMUM VALUE = 5 IN LOCATION 4

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

CHAPTER 6

The Compiler Writing System

6.1 Overview

This chapter presents an introduction to the compiler writing system employed in the PL/STAR project. The system was developed by Professor Donegan of the College of William and Mary and adapted for use on the CDC machines at NASA by the author.

The system, itself written in PASCAL, generates the basis of a syntax directed compiler for a target language defined by a LALR(1) grammar. The generated compiler is also a PASCAL program.

Certain modifications to the original system were made at NASA to substantially reduce the amount of user interaction needed.

The system proper consists of three programs, a table generator, a table compressor and a compiler synthesizing program. Also available is a utility program that puts the grammar rules and semantic code of an existing version of the compiler into a form that can be used as input to the compiler writing system.

6.2 Input/Output

Two files are input to the system. The first contains the grammar for the language. Following each grammar rule is the code to execute semantic actions associated with the rule.

The form of an input rule resembles BNF with a few modifications. The ' ::= ' is replaced by a ':', and the vertical or bar is replaced by a semicolon. Symbols on the right hand side of a rule are separated by commas and each rule is terminated by a period. A '\$' must precede each ':', ';', ',' and '.' appearing in a rule.

The end of the grammar is signaled by a period in column one.

The code associated with a rule appears in the output compiler following the case label corresponding to the rule in the SYNTHESIZE case statement. Line images containing semantic actions must have a blank in column one while rules must not have a blank there.

The output of the system is the parse tables and the generated compiler. In addition various human readable lists, such as lists of symbols and state tables, that aid in debugging are printed.

6.3 Method

The table generator builds the LR(0) sets of items for the grammar and adds lookahead information for those states with parsing conflicts. The generator assigns an integer index to each symbol in the grammar. These indexes represent the symbols in the tables.

The table compressor attempts to shrink the size of the tables.

The compiler synthesizing program merges textfiles supplied by the user and the table generator program into a skeleton compiler for the target language. The completeness of the generated compiler depends upon the sophistication of the user supplied code.

The generated compiler is written to a segmented file called BUILDFL by the compiler writing system.

The first segment of the compiler is created by the table generator program. The segment contains the program card for the compiler and certain constant declarations.

Two additional constant declarations are contained in the second segment, supplied by the compressor program.

To this base the compiler synthesizer adds the first of two sections of user supplied code. This first section contains global declarations and the function and procedure declarations that are to appear before procedure SYNTHESIZE, the semantic action driver.

The next file to be merged is supplied by the table generator. It contains procedures SCANINIT, which equates reserved words with their integer indices, NEXTSYM, which identifies the next symbol in the input to the compiler and SYNTHESIZE.

The last segment of BUILDFL contains additional user supplied code and the two system routines, READTABS and PARSE, that execute the parse.

CHAPTER 7

Evaluation

In [9] Pratt lists features of a good programming language. This chapter attempts to evaluate PL/STAR using Pratt's criteria. PL/STAR, though, does not exist in a vacuum. It is an alternative to the STAR assembly language and should be judged accordingly.

7.1 Clarity, Simplicity and Unity of Language Concept

Pratt feels that a language should provide its user a coherent and manageable set of concepts with which to develop programs.

PL/STAR maintains a small number of native forms. The rules governing their use will be familiar to assembly language programmers.

The PL/STAR language, though, is designed to be used as

a basis for development and, therefore the features provided are designed to support a wide variety of applications.

7.2 Clarity of Program Structure

It is in this area that PL/STAR contrasts most strongly with assembly language. The control structures of PL/STAR can make clear the flow of the program logic. The free form input allows the use of blank lines to delimit code of equal precedence while indentation can show subordinate or dependent code. These features properly used result in a piece of work more nearly resembling the human concept of the process of the module than the machine's linear instruction sequence.

7.3 Ease of Extension

Pratt means by ease of extension a measure of how well the algorithm can be expressed using the features available in the language. To address this concern, PL/STAR contains the structured types array and record, which permit the user to build his own data structures.

7.4 Efficiency

Pratt mentions three distinct aspects of efficiency.

7.4.1 Efficiency of Program Execution

In high level languages the generation of the object code is hidden from the programmer. A result of the close correspondence of PL/STAR to machine features is that the PL/STAR programmer controls the nature of the code that is generated. The efficiency of the code, then, depends greatly upon the skill of the programmer and is expected to approximate that of assembler code written by the same programmer.

7.4.2 Efficiency of Program Translation

The PL/STAR compiler exists on the STAR access station, a CDC 6400 computer. After PL/STAR programs compile on the access station the object files are sent to the STAR for loading and execution.

STAR assembler code, though, is assembled directly on the STAR, making any comparison of the two times misleading. Certainly a CPU second does not represent the same expenditure of resources on the access station as on the

STAR.

7.4.3 Efficiency of Program Creation, Testing and Use

In the sense that efficiency is related to ease, this topic summarizes the purpose of the language. PL/STAR is an attempt to bring high level features to assembly level programming. It is reasonable to expect that some portion of the relative advantage that high level languages enjoy over assembler in program development time will accrue also to PL/STAR.

APPENDIX

The PL/STAR Grammar

This appendix lists the grammar that defines PL/STAR. It is the grammar used as input to the compiler writing system except that it appears below in standard BNF. The modifications to BNF necessary to make grammars acceptable input to the compiler writing system are contained in chapter 6.

```
[ 1] <compilation> ::= <moduledef> <eofsym>
[ 2] <moduledef>   ::= <modulehead> <constsection>
    <typesection> <declarations> <initialsection>
    <codesection> ;

[ 3] <modulehead> ::= MODULE <identifier> ;

[ 4] <constsection> ::= CONST <constlist>
[ 5] <constsection> ::= <epsilon>
[ 6] <constlist>   ::= <constelem>
[ 7] <constlist>   ::= <constelem> <constlist>
[ 8] <constelem>   ::= <identifier> = <constant> ;

[ 9] <typesection> ::= TYPE <typelist>
[10] <typesection> ::= <epsilon>
```

```

[ 11] <typelist>      ::= <typedef> <typelist>
[ 12] <typelist>      ::= <typedef>
[ 13] <typedef>       ::= <typeid> = <recorddef>
[ 14] <typeid>        ::= <identifier>

[ 15] <recorddef>     ::= <record> <recdeflist> ENDR ;
[ 16] <record>        ::= RECORD
[ 17] <recdeflist>    ::= <recdefelem> <recdeflist>
[ 18] <recdeflist>    ::= <recdefelem>
[ 19] <recdefelem>    ::= <fieldidlist> : <predeftype> ;
[ 20] <fieldidlist>   ::= <fieldidhead> <fieldidlist>
[ 21] <fieldidlist>   ::= <identifier>
[ 22] <fieldidhead>   ::= <identifier> ,

[ 23] <generaltype>   ::= <usertype>
[ 24] <generaltype>   ::= <predeftype>
[ 25] <usertype>      ::= <identifier>
[ 26] <predeftype>    ::= REAL
[ 27] <predeftype>    ::= INTEGER
[ 28] <predeftype>    ::= DESCRIPTOR
[ 29] <predeftype>    ::= CHARACTER
[ 30] <predeftype>    ::= BIT

[ 31] <declarations>  ::= VAR <decllist>
[ 32] <decllist>      ::= <declelem> <decllist>
[ 33] <decllist>      ::= <declelem>
[ 34] <declelem>      ::= <identifierlist> : ARRAY [
    <bound> ] OF <predeftype> ;
[ 35] <bound>         ::= <constant>
[ 36] <declelem>      ::= <identifierlist> : <predeftype>
    <storage> ;
[ 37] <storage>       ::= SREG
[ 38] <storage>       ::= DREG
[ 39] <storage>       ::= SDREG
[ 40] <storage>       ::= REG
[ 41] <declelem>      ::= <identifierlist> : <generaltype>
    ;
[ 42] <declelem>      ::= <identifierlist> : <record>
    <recdeflist> ENDR ;
[ 43] <identifierlist> ::= <idlisthead> <identifierlist>
[ 44] <identifierlist> ::= <identifier>
[ 45] <idlisthead>    ::= <identifier> ,

[ 46] <initialsection> ::= VALUE <initlist>
[ 47] <initialsection> ::= <epsilon>
[ 48] <initlist>      ::= <initelem> <initlist>
[ 49] <initlist>      ::= <initelem>
[ 50] <initelem>      ::= <identifier> = <constant>

[ 51] <codesection>   ::= CODE <statementlist> ENDM
[ 52] <statementlist> ::= <statement> ; <statementlist>
[ 53] <statementlist> ::= <statement>

```



```

[ 54] <statement>      ::= <label> <unlabstatement>
[ 55] <statement>      ::= <unlabstatement>
[ 56] <label>          ::= <identifier> :

[ 57] <unlabstatement> ::= <assignmentstate>
[ 58] <unlabstatement> ::= GOTO <identifier>
[ 59] <unlabstatement> ::= ENTRY <entryplist>
[ 60] <unlabstatement> ::= ENTRY
[ 61] <unlabstatement> ::= CALL <callid> <callplist>
[ 62] <unlabstatement> ::= CALL <callid>
[ 63] <unlabstatement> ::= <instruction>
[ 64] <unlabstatement> ::= <epsilon>
[ 65] <unlabstatement> ::= <structuredstatement>

[ 66] <assignmentstate> ::= <lhs> <rhs>
[ 67] <lhs>              ::= <identifier> <becomes>
[ 68] <lhs>              ::= <identifier> ^ <becomes>
[ 69] <becomes>          ::= :=
[ 70] <rhs>              ::= <operand1>
[ 71] <rhs>              ::= <operand1> ^
[ 72] <rhs>              ::= <constant>
[ 73] <rhs>              ::= <operand1> <boolop> <operand2>
[ 74] <rhs>              ::= <operand2> <arithop> <operand2arith>
[ 75] <rhs>              ::= LENGTH ( <identifier> )
[ 76] <operand1>         ::= <identifier>
[ 77] <operand2>         ::= <identifier>
[ 78] <operand2arith>    ::= <identifier>
[ 79] <operand2arith>    ::= <constant>
[ 80] <boolop>          ::= AND
[ 81] <boolop>          ::= OR
[ 82] <boolop>          ::= XOR
[ 83] <arithop>         ::= <addop>
[ 84] <arithop>         ::= <mulop>
[ 85] <addop>           ::= +
[ 86] <addop>           ::= -
[ 87] <mulop>           ::= *
[ 88] <mulop>           ::= /

[ 89] <entryplist>      ::= <entryplfront> <entrylist>
   <entryplback>
[ 90] <entryplfront>    ::= (
[ 91] <entrylist>       ::= <regpair> , <entrylist>
[ 92] <entrylist>       ::= <regpair>
[ 93] <entryplback>    ::= )
[ 94] <regpair>         ::= <identifier>
[ 95] <regpair>         ::= <epsilon>

[ 96] <callid>          ::= <identifier>
[ 97] <callplist>       ::= <callplfront> <callplback>
[ 98] <callplfront>     ::= ( <callplitem>
[ 99] <callplback>      ::= , <callplitem> <callplback>
[100] <callplback>     ::= )

```

```

[101] <callplitem> ::= <constant>
[102] <callplitem> ::= <callregid>
[103] <callregid> ::= <identifier>
[104] <callregid> ::= [ <identifier> ]

[105] <instruction> ::= (

[106] <structuredstatement> ::= <ifstatement>
[107] <structuredstatement> ::= <whilestatement>
[108] <structuredstatement> ::= <repeatstatement>

[109] <condition> ::= <operand1> <relop> <operand2>
[110] <relop> ::= <
[111] <relop> ::= >
[112] <relop> ::= =
[113] <relop> ::= <>
[114] <relop> ::= <=
[115] <relop> ::= >=

[116] <ifstatement> ::= <if> <condition> <then>
<statementlist> ENDI
[117] <ifstatement> ::= <if> <condition> <>truepart>
<statementlist> ENDI
[118] <if> ::= IF
[119] <then> ::= THEN
[120] <>truepart> ::= <then> <statementlist> ELSE

[121] <whilestatement> ::= <whilehead> <statementlist> ENDW
[122] <whilehead> ::= WHILE <condition> DO

[123] <repeatstatement> ::= <repeat> <statementlist>
<untilpart>
[124] <repeat> ::= REPEAT
[125] <untilpart> ::= UNTIL <condition>

```

Notes

- [1] Control Data Corporation, Control Data STAR-100 Computer Hardware Reference Manual (Control Data Corporation, 1974) p. 1-1.
- [2] NASA Langley Research Center, STAR Programing Manual (NASA Langley Research Center, 1976) sec. 1.2.1, 1.2.2.
- [3] Niklaus Wirth, "PL360, A Programming Language for the 360 Computers," Journal of the Association for Computing Machinery 15 (January 1968) : 37-74.
- [4] Victor S. Foster, "MIDAS: A Mid-level Language for Microprocessors," (University of Virginia, undated).
- [5] Uwe Frederik Pleban, "Design and Implementation of the Structured Assembly Language PL/85" (Master's Thesis, University of Kansas, 1976).
- [6] NASA Langley Research Center, SL/1 Manual, (NASA Langley Research Center, 1978).
- [7] Kathleen Jensen and Niklaus Wirth, Pascal User Manual and Report (New York: Springer Verlag, 1974).
- [8] NASA Langley Research Center, STAR Standard Procedures, (NASA Langley Research Center, 1976) pp. 28, 29.
- [9] Control Data Corporation, STAR Operating System Version 1 (Control Data Corporation, 1977) p. 8-1 - 8-11.
- [10] Terrence W. Pratt, Programming Language Implementation and Design (Englewood Cliffs, NJ: Prentice-Hall, 1975), pp. 6-10.

References

Aho, Alfred V., and Ullman, Jeffrey D. Principles of Compiler Design. Reading, Mass.: Addison-Wesley, 1977.

Aho, Alfred V., and Ullman, Jeffrey D. The Theory of Parsing, Translation and Compiling, 2 vols. Englewood Cliffs: Prentice-Hall, 1972.

Control Data Corporation. Control Data STAR-100 Computer Hardware Reference Manual. Arden Hills, Minn: Control Data Corporation, 1974.

Control Data Corporation. Implementation Language (IMPL) Reference Manual. Sunnyvale, Calif.: Control Data Corporation, 1973.

Control Data Corporation. STAR Operating System Version 1 Reference Manual. Sunnyvale, Calif.: Control Data Corporation, 1977.

Foster, Victor S. "MIDAS: A Mid-level Language for Microprocessors." University of Virginia, undated.

Jensen, Kathleen and Wirth, Niklaus. Pascal User Manual and Report. New York: Springer Verlag, 1974.

NASA Langley Research Center. SL/1 Manual. Hampton, Va: NASA Langley Research Center: 1978.

NASA Langley Research Center. STAR Programing Manual. Hampton, Va: NASA Langley Research Center, 1976.

NASA Langley Research Center. STAR Standard Procedures. Hampton, Va: NASA Langley Research Center, 1976.

Pleban, Uwe F. "Design and Implementation of the Structured Assembly Language PL/85." Masters Thesis, University of Kansas, 1976.

Pratt, Terrence, W. Programming Language Implementation and Design. Englewood Cliffs, NJ: Prentice-Hall, 1975.

van der Poel, W. L. and Maarssen, L. A. eds. Machine Oriented Higher Level Languages: Proceedings of the IFIP Working Conference on Machine Oriented Higher Level Languages. Amsterdam: North Holland Publishing Company, 1974.

Wirth, Niklaus. "PL360, A Programming Language for the 360 Computers." Journal of the Association for Computing Machinery 15 (January 1968) : 37-74.

VITA

BARRY LEE MOWDAY

Born in Coatesville, Pennsylvania December 7, 1952. Graduated from Coatesville Area Senior High School in that city in June 1970.

Received B.A. degree College of William and Mary, June 1974. In August 1977, the author entered the program in Applied Science at the College of William and Mary. Since that time he has served as a research assistant in the Programming Techniques Branch of the NASA Langley Research Center in Hampton, Virginia.