c.1

A TRANSLATOR WRITING SYSTEM FOR MINICOMPUTERS

by

Jake Madderom
B.A.Sc., University of British Columbia, 1968

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in the Department
of
Electrical Engineering

We accept this thesis as conforming to the
required standard

THE UNIVERSITY OF BRITISH COLUMBIA

June 1973

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical Engineering

The University of British Columbia
Vancouver 8, Canada

Date June 8, 1973

ABSTRACT

Some portions of real-time computer process control software can be programmed with special purpose high-level languages. A translator writing system for minicomputers is developed to aid in writing translators for those languages. The translator writing system uses an LR(1) grammar analyzer with an LR(1) skeleton parser. XPL is used as the source language for the semantics. An XPL to intermediate language translator has been written to aid in the translation of XPL programs to minicomputer assembly language. A simple macro generator must be written to translate intermediate language programs into various minicomputer assembly languages.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

GLOSSARY

BNF - Backus-Naur Form. A metalanguage used in writing grammar definitions.

Byte - Eight binary bits.

Canonical derivation - The derivation in which only the right non-terminal is replaced at each step.

Canonical parse - The reverse of the sequence of phrases that were used in the canonical derivation.

Configuration - The states of the LR(1) parser consist of configuration sets. A configuration is a production with one symbol marked as a successor.

Context-free phrase-structure grammar - A phrase-structure grammar having only a single non-terminal symbol as the left part of the productions of the grammar.

Derivation - A derivation of a string is the process of substituting phrases for non-terminal symbols, starting with the goal symbol, until the string is obtained.

Extended precedence grammar - Those grammars for which a unique precedence relationship exists between all legal symbol pairs and the next symbol.

FSM - Finite State Machine. The LR(1) parser is a finite state machine.

Goal Production - The production that defines the goal symbol.

Goal symbol - The non-terminal symbol that does not appear in the right part of any production.

Grammar - Set of rules, called productions, that determine which strings are valid.

Inadequate state (of an LR(1) parser) - States with two or more reductions or states with both transition(s) and reduction(s).

Look-ahead set - A set of terminal symbols used by LR(k) parsers to make stacking decisions.

LR(k) grammar - A grammar whose programs may be parsed during a single left to right scan without looking ahead more than k symbols.

Macro - A series of assembly language instructions.

Metalanguage - A language used to define another language.

Metasymbols - The symbols of a metalanguage.

MINCOM - The name given to the XPL to intermediate language translator written for this thesis.

MSP(2,1;1,1) - Mixed Strategy Precedence(2,1;1,1). The pair 2,1 refers to the two symbols in the parse stack and the single next symbol used to make precedence stacking decisions. The 1,1 pair refers to the number of symbols to the left and right of a proposed reduction used to check context.

Non-terminal symbol - A name given to a phrase.

Nova - A Data General minicomputer.

Parse - A parse of a string indicates which phrases were used to form the string. All programs must reduce to the goal symbol.

Parser - The portion of a translator that recognizes series of terminal and non-terminal symbols as phrases.

Phrase - An ordered sequence of terminal and non-terminal symbols.

Phrase-structure grammar - A grammar indicating all valid phrases.

PLALR(1) - Parser Look-Ahead Left to Right one. A parser which calculates the look-ahead sets when needed rather than using stored look-ahead sets was written for this thesis.

Production - A rule of a grammar.

Program - A valid string.

Programming language - A set of programs.

Reduction - The replacement of a sequence of symbols with an equivalent non-terminal symbol.

Reserved word - A terminal symbol in a grammar definition that is not an ‹identifier›, ‹string›, ‹number› or special character.

Scanner - The portion of a translator that reads the input text and recognizes the terminal symbols.

Semantic routine - The portion of a translator that does all the data processing of input data.

Simple precedence grammar - Those grammars for which a unique precedence relationship exists between each possible symbol pair.

SLR(1) - Simple left to Right one. A grammar is SLR(1) if it is LR(1) and all the look-ahead sets associated with an inadequate state are disjoint.

String - A sequence of terminal symbols.

Successor - A successor symbol in a state (referring to the states of an LR(1) parser) is a symbol that may follow a state. The successor has associated with it a next state.

System Builder - Special purpose translator program.

Terminal symbol - A symbol in a grammar definition that does not appear as a left part of a production.

Transition (referring to LR(1) parsers) - A configuration with a non-null successor symbol.

TWS - Translator Writing System.

UNCOL - Universal Computer Oriented Language.

Vocabulary - The set of terminal symbols of a programming language.

VSLR(1) grammar - Very Simple Left to Right one grammar. A grammar is VSLR(1) if it is LR(1) and each state has zero or one reduction.

XPL - The name of a programming language. A dialect of the language PL/1.

ACKNOWLEDGEMENT

I wish to express my gratitude to the people who have assisted me during this research project. I am deeply grateful to Dr. J.S. MacDonald, my research supervisor, for his helpful suggestions, encouragement and time. I appreciate the interest Dr. G.F. Schrack showed in my research and thank him for his valuable comments.

Thanks are also due to Mr. Peter Madderom for his help with proof reading and to Miss Norma Duggan for typing the final manuscript.

A special thanks to my wife, Marie, for her encouragement and understanding during my graduate program.

I am grateful to the National Research Council of Canada and to the University of British Columbia for their financial support.

1. INTRODUCTION

The purpose of this thesis is to develop a translator writing system (TWS) for minicomputers. The TWS is to consist of programs which run on a large computer and which produce translators written in a specified minicomputer machine code or assembly language. The ideal configuration for the TWS is shown in Figure 1.1. The actual configuration produced for this thesis is discussed in Chapter 2. The original concept of a TWS for minicomputers came out of MacDonald Dettwiler and Associates' (MDA) original efforts with a translator for the B.C. Telephone portion of the Northern Interprovincial Radio System (NIPRS) [1]. The grammar for the B.C. Telephone translator (called a system builder by MDA) is given in Appendix 1.

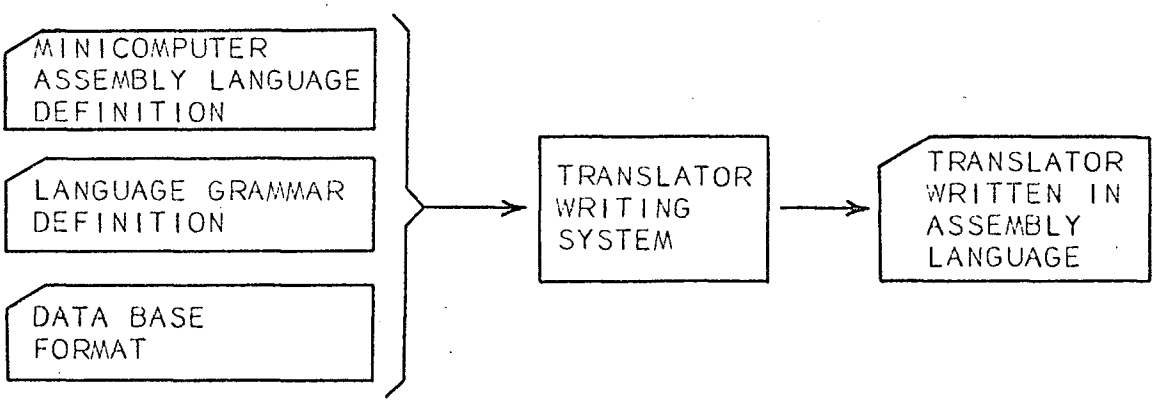


Figure 1.1 Ideal TWS

The software for the NIPRS supervisory system consisted of two distinct parts:

1. The operating system, consisting of a supervisor, interrupt

handler, device handlers, logic interpreter, and other basic tasks. These programs were coded in assembly language by MDA.

2. The system definition file, consisting of messages, description of input and output signals, and logic equations for reporting and control. This data was defined by B.C. Telephone personnel using the system builder.

The system builder was a translator that converted English statements into the system definition file. The system builder ran on the spare supervisory minicomputer. Because the input language to the system builder was very high-level and was designed to correspond to the normal jargon of the telephone industry, B.C. Telephone personnel were able to program their own system in a fraction of the time that would have been required using a general purpose language. Special purpose languages such as AUTRAN, BATCH, IBM PROSPRO/1800, and G.E.'s BICEPS [2] have been developed to enable users of process control systems to write their own software without having to worry about the structure of the software system or the computer hardware. These languages served basically the same purpose as the B.C. Telephone system builder but were written for general process control applications and could not take advantage of special industry jargon or hardware configurations.

Creating a special high-level language for each supervisory or control application requires a different translator (system builder) program for each application. The translator should run on a computer that is readily available to the user since most system changes require modifications to the system definition file. That computer could be a spare control minicomputer or a large data processing computer. A translator writing system would reduce the programming time required to

produce special purpose high-level language translators. Since translator writing systems for large computers such as the IBM 360 series exist, this thesis deals mainly with a TWS for minicomputers.

The programs that form the TWS were written to run on the IBM 360/67 available at UBC. The TWS can be used on any large IBM 360 or 370. A Data General Nova minicomputer with a four thousand word memory was used as the target computer for the sample translators produced in this thesis. Only the portions of the translators produced by the TWS were run on the Nova. Complete translators require more than four thousand words of memory.

The thesis is organized as follows. Chapter 2 gives an outline of all the components of a minicomputer TWS. Chapter 3 presents some background material on grammars and various types of parsing algorithms. The high-level language XPL [3] and the compiler required to translate XPL into minicomputer assembly language is described in Chapters 4 and 5. Various parsing routines are tried and compared in this thesis. The grammar analyzer algorithms and their associated parsers are presented in Chapter 6. Chapter 7 gives a brief summary of some of the error recovery techniques attempted. Chapter 8 compares the table sizes for different grammars and discusses an example of a translator implemented on the Nova. Chapter 9 provides the conclusions and indicates what further work is required.

2. A TRANSLATOR WRITING SYSTEM FOR MINICOMPUTERS

The structure of the translator writing system developed in this thesis is identical to that of the XPL TWS of McKeeman et al. [3]. XPL is the name of a dialect of PL/1 developed by McKeeman specifically for translator writing. There are four programs in the XPL TWS. A grammar analyzer is used to translate grammars written in relaxed Backus-Naur Form (BNF) [4] into XPL declaration statements. These declaration statements are combined with a skeleton translator to form a parser for the grammar. User written semantic routines must be combined with the parser to form a complete translator. The other two programs in the XPL TWS are XCOM, an XPL to IBM 360 machine code compiler, and *XPL, an assembly language monitor. The monitor interfaces with the operating system so that only the monitor need be changed to use the XPL TWS with any 360 operating system.

A minicomputer TWS could have been produced by using the XPL TWS and changing XCOM to a minicomputer machine code compiler. That approach was not followed for two reasons. First, to be useful for real-time systems, a TWS must be target machine independent. A compiler would be restricted to one minicomputer. Secondly, the grammar analyzer of the XPL TWS accepts a limited class of grammars and produces tables of size proportional to the square of the number of symbols in the language.

An XPL to intermediate language translator was developed to make the TWS target machine independent. A macro generator must be written to convert the intermediate language to a particular minicomputer machine code or assembly language. A new grammar analyzer

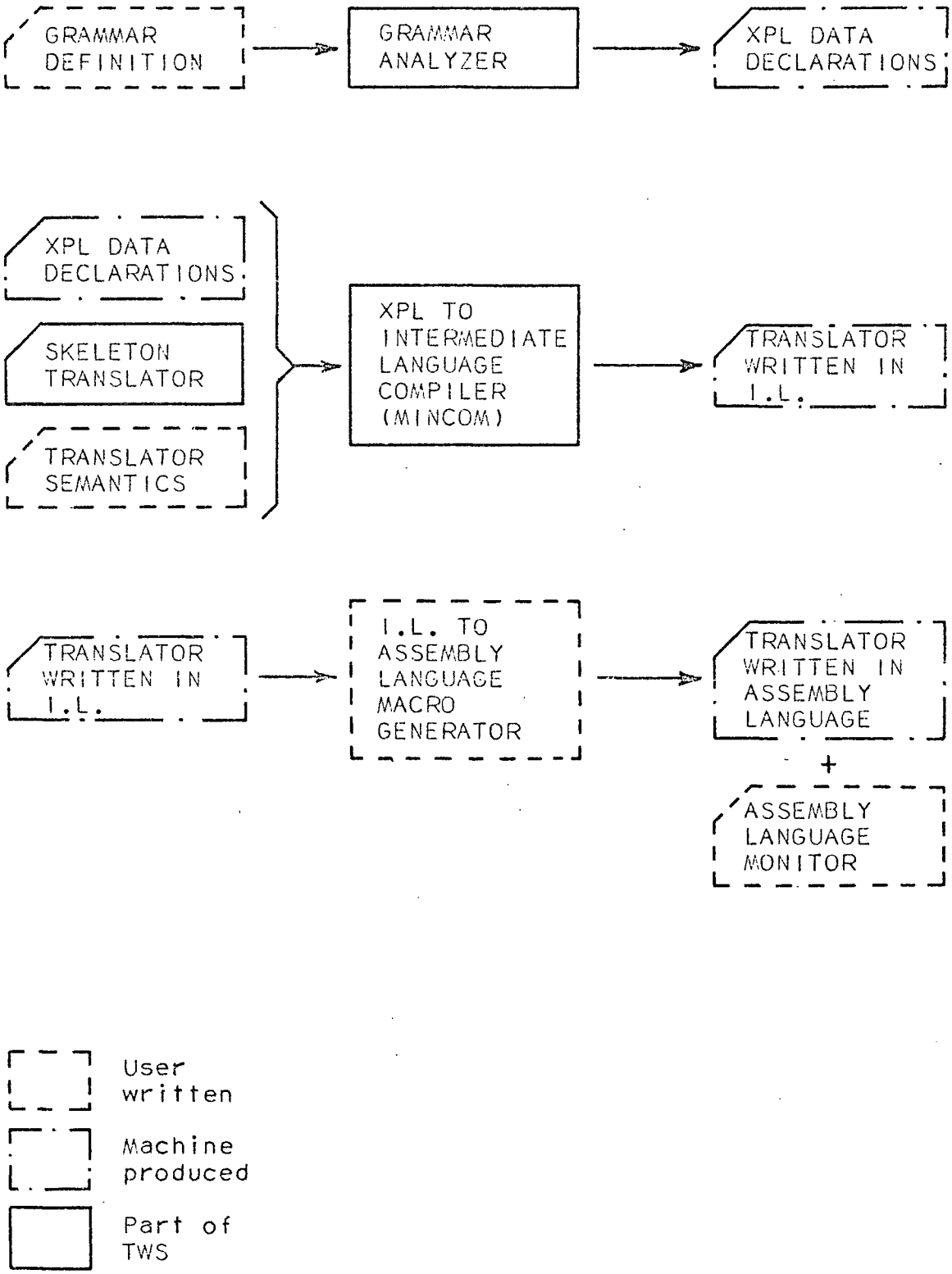


Figure 2.1 Block Diagram of Minicomputer TWS

and skeleton translator were written in an attempt to extend the class of acceptable grammars and to reduce the size of the translator tables. The language XPL was retained as the high level language for the semantic routines.

A block diagram for the minicomputer translator writing system is given in Figure 2.1.

The skeleton translators discussed in this thesis consist of three main subroutines: a parser, a scanner, and an error recovery routine. The scanner was the same for all the translators produced. The scanner is called by the parser whenever the parser requires another symbol. The scanner reads the input text and searches for predefined symbols. It is able to recognize all the special single characters and reserved words defined in the grammar and the symbols <idenfier>, <string> and <number>.

An <identifier> is defined within the scanner as any sequence of characters beginning with an alphabetic character and containing only alphabetic characters and numbers. A <number> is any sequence of the characters 0 to 9. A <string> is any sequence of characters beginning with an apostrophe and ending with an apostrophe or carriage return. Symbols may not be continued on the following line.

The symbols <identifier>, <string> and <number> are defined in the translator at run time by initializing variables. Those variables may be changed by the semantic routines. If, for instance, at some point in the translation, blanks must be included within an <identifier> then a blank may be defined as an alphabetic character (leading blanks would still be ignored).

The parser determines which sequence of productions must be

applied to reduce the input text to the final goal symbol (such as <program>). Each time the parser recognizes a production, the semantic routine associated with that production is called.

The error recovery routines developed in this thesis served two purposes. The first was to recover from syntax errors. When the parser recognizes a syntax error, some attempt must be made to parse the rest of the input correctly. The second purpose of the recovery routine was to enable reserved words to be used as <identifier>s. The scanner recognizes all reserved words. If a reserved word is used out of its correct context, the recovery routine changes the symbol number to that of an <identifier> and restarts the parse. This technique allows all reserved words to be used as <identifier>s in all contexts where the reserved word is illegal.

3. GRAMMARS

Grammars expressed in BNF are not necessarily acceptable to a translator writing system. This chapter outlines some of the types of grammars which have been used in this thesis. Before grammars can be discussed, a few definitions must be given.

The character sequences recognized by the scanner will be called terminal symbols or symbols. The symbols form a vocabulary. A string is any sequence of these symbols. A grammar is a set of rules, called productions, that determine which strings are valid. A program is any valid string. A programming language is a vocabulary, a grammar, and a set of procedures which define the meaning of the rules of the grammar.

An ordered group of symbols is called a phrase. A non-terminal symbol is a name given to a phrase. A phrase may consist of non-terminal symbols as well as terminal symbols. A phrase-structure grammar for a language indicates all valid phrases. A context-free phrase-structure grammar is a phrase-structure grammar having only a single non-terminal symbol as the left part of the productions of the grammar.

The term "context-free" is a misnomer. It does not mean that the symbols forming a string do not change meaning with context. It only indicates that, beginning with a unique non-terminal called a goal symbol, substituting any phrase for a non-terminal symbol until there are no more non-terminal symbols will result in a program. The phrase that may be substituted for a non-terminal is not dependent on the symbols surrounding the non-terminal.

There must be only one production in a grammar with a left part that does not appear in any other production. That production is called the goal production. The right part of the goal production must contain at least one non-terminal symbol otherwise the language will consist of a single string. A derivation of a string is the process of substituting phrases for non-terminals. A canonical derivation is a derivation in which only the rightmost non-terminal is replaced at each step. A canonical parse of a string is the reverse of the sequence of productions that were used in the canonical derivation. A parse of a string indicates which productions were used to produce the string.

3.1 Parsing Algorithms

One way of implementing parsing algorithms is through the use of parse stacks and constant tables. A one-pass parser uses the information in the tables, the information in the parse stack and the next k symbols from the input text to determine how to proceed with the parse. The parse stacks contain a past history of the parse and the tables describe the grammar.

The parser does one of the following:

1. It can reduce the stack. A reduction is the replacement of a sequence of symbols on the parse stack with a non-terminal symbol. If it makes a reduction then a production has been recognized and a semantic routine associated with that production is executed.

2. It can recognize that an error has occurred and can make some attempt at recovery.

3. It can stack the next symbol.

4. It can ask for another symbol from the scanner before doing one of the other three.

3.2 Types of Context-free Grammars

This thesis discusses simple precedence grammars, extended precedence grammars and left to right one (LR(1)) grammars. A simple precedence grammar is one for which a unique precedence relationship exists between each possible symbol pair. An extended precedence grammar is one for which a unique precedence relationship exists between all legal symbol pairs and the next symbol. An LR(1) grammar is one whose programs may be parsed during a single left to right scan without looking ahead more than one symbol.

The simple precedence grammar parsers use only the top parse stack symbol and the next input symbol to decide whether to reduce, stack, or call an error routine. A simple two dimensional array is sufficient to indicate one of the above three actions for every symbol pair in the grammar. The decision to reduce alone is not enough. The parser must determine which production is applicable. With the precedence grammars, that can be accomplished by comparing the top symbols in the stack with the right parts of the productions.

The extended precedence grammar parsers use the top two symbols on the parse stack with the next input symbol to make the reduce, stack, or error decision. To store all possible triple combination of symbols requires a three-dimensional array that is usually quite sparse because most triples cannot occur in practical languages. McKeeman et al. [3] reduced the dimension of the array required by adding a check-triple type to the two-dimensional simple precedence array. Only legal triples need be stored or checked.

The above two grammars severely limit the way the BNF

productions can be written. Very little context is actually used in the parse. One extension to the precedence approach was made by McKeeman et al. [3]. Their Mixed Strategy Precedent Grammar (MSP(2,1;1,1)) uses the two symbols on the stack top with the next input symbol (2,1) to make most parsing decisions. The (1,1) pair refers to a single symbol to the left or right of a production checked when left or right context is required.

LR(1) grammars form a more general class of grammars than the simple and extended precedence grammars. All simple and extended precedence grammars are LR(1). Knuth [5] showed that a parsing algorithm for an LR(k) grammar is in one of N states during a parse. Each state contains information relating the next input symbol to a following state or a reduction. Reductions force the parser into a previous state.

Chapter 6 describes how the above three types of grammars were used to implement parsers for the minicomputer TWS.

4. SEMANTICS

Chapter 3 described various parsers that can be used for phrase-structure grammars. The parser, combined with a scanner, accepts a user input program and recognizes when to apply the productions of the grammar. The parser does not know the meaning associated with each production. The parser accepts symbols from the scanner and stacks the symbols until a production is recognized. It then calls a semantic routine associated with the production before reducing the stack.

The semantics must associate a series of machine language instructions with each production. The changes that these machine language instructions make to any output medium that the translator uses (magnetic tape, disc, core buffer, teletype etc). define the semantics of a production. Expressed in this way, semantics do not indicate some abstract meaning, they only indicate actions that must be performed by the computer when a production is recognized by the parser.

One way to indicate the semantics of a production would be to express the semantics as a series of machine instructions. That would produce an efficient translator but would require a large amount of programming time. It would also force a complete rewrite of the semantic routines every time a translator was required for a different computer.

Since the translator is not a real-time program, the execution efficiency is not an important consideration, and a high-level programming language may be used for semantic routines. FORTRAN was an obvious choice for minicomputers because it has been implemented on most available machines. But FORTRAN does not have the string handling capabilities or the data types that must be used in manipulating a complex data base. FORTRAN was not used because a language with all the required features existed.

4.1 XPL

McKeeman et al [3] defined XPL, a dialect of PL/1, in order to create a language that could be used for writing translators. Since XPL contains all of the features required for data manipulation, XPL was used as the high level language for the semantic routines. The decision to use a language other than FORTRAN made it necessary to incorporate a compiler into the TWS. That compiler had to translate XPL programs into the assembly language of a target minicomputer. A two stage translation process was used. The program MINCOM was written to translate XPL programs to an intermediate language. McKeeman's translator writing system was used to produce MINCOM. A macro generator was used to translate the intermediate language to a minicomputer assembly language.

The complete grammar for XPL as used in this thesis is given in Appendix II. Two changes were made to McKeeman's definition in order to decrease the storage requirements of the translator. Dynamic

storage allocation was used to implement variable length character strings on the IBM 360. Although dynamic storage allocation does make efficient use of core, the routines used to assign and release core occupy a significant amount of space. The COMPACTIFY routine of the XPL skeleton translator requires eight hundred bytes of memory.

The XPL programs call a monitor for all input, output and string manipulation. The monitor written for the Nova computer occupied about seven hundred 16 bit words. The monitor must be written in assembly language and must be rewritten every time XPL is implemented on a new computer. Since the monitor does all string manipulation, the dynamic string assignment routines would have to be part of the monitor and would have to be written in assembly language. Dynamic string allocation would force more interdependence between the various string manipulation routines and make them more difficult to write.

Dynamic string allocation is not part of the XPL definition. It is only the way variable length strings were implemented on the IBM 360. Variable length strings are essential for easy string manipulation and must be implemented. One solution would be to assign a maximum buffer area for each string, but that would require too much memory space. A change was made to the XPL grammar to allow a maximum length to be specified for any string. If no length is specified, a default length is assumed. MINCOM allows any length to be specified. Strings longer than 256 characters may be used to store string constants. The individual constants may be accessed with a substring routine. One other change was made to the XPL grammar to allow more efficient use of subroutine variable storage. The IBM 360 implementation assigned permanent storage to all bit and fixed data types declared within a

procedure. Since not all procedures call all other procedures, that storage could be shared. The MINCOM parser uses only a single left to right scan on the input program and cannot determine which procedures are going to be called from within the current procedure. MINCOM must know which procedures are to be called in order to assign storage that will not be used by those procedures. The data type NESTED was added to the grammar to convey the above information to MINCOM.

The semantic routines are associated with the appropriate production via a CASE statement. The parser, when it has determined that a reduction is necessary, calls the procedure SYNTHESIZE with the production number as a parameter. Control is returned to the parser when the appropriate semantic routine is finished.

4.2 Bit Data Types

MINCOM allows the length of the data type BIT to vary in length from one to the word length of the minicomputer. Single BIT variables are assigned a full word of memory. Bit arrays are assigned only as much storage as is required. For example, the statement

```
DECLARE A(15)  BIT(1);
```

would cause a single 16 bit word to be assigned to the array A.

5. MINCOM - AN XPL TO INTERMEDIATE LANGUAGE COMPILER

In June 1958, System Development Corporation started to investigate procedure-oriented programming language concept for real-time applications [7]. They developed JOVIAL - a programming language for real-time command systems. They used a Generator (written in JOVIAL) to translate a JOVIAL program into a Universal Computer Oriented Language (UNCOL). A Translator (also written in JOVIAL) was used to transform UNCOL into assembler. Only the translator needed to be changed in order to implement JOVIAL on another computer.

A similar approach as that used in the implementation of JOVIAL was used in this thesis. MINCOM is a compiler that translates an XPL program into an intermediate language. The intermediate language was designed to be well suited for the translation of XPL to minicomputer assembly languages. MINCOM uses a set of constants which describes the target minicomputer assembly languages. Constraints such as word-length, op-code types, number of general registers, number of index registers and types of addressing may be changed by changing constants in the MINCOM source. The intermediate language to assembly language translator is a very simple macro generator.

There have been other attempts to define a Universal Computer Oriented Language (UNCOL) on an assembly language level [8]. Such a definition would enable all programs to be expressed in UNCOL and would eliminate the reprogramming costs associated with changing computers. This approach has not been successful in the past because a UNCOL could never incorporate all the special instructions of all computers.

UNCOL programs could not make efficient use of any particular computer.

The TWS described in this thesis is meant to create programs to be run on spare control computers. The memory size may be as small as eight thousand words but the program execution efficiency need not be optimum. The concept of UNCOL has been implemented but the language used has been termed an intermediate language rather than a Universal Computer Oriented Language because intermediate language programs can be used only if the execution speed of that program is of minor concern and the source language is XPL. There is no universal use implied.

The term minicomputer needs some explanation. MINCOM has been written to minimize the size of the intermediate language program. Only for minicomputers is the cost of memory size more important than the execution time of the program. The intermediate language has been defined to make use of a very limited instruction set. Only for computers with a short word length (and a small instruction set) will reasonably efficient programs result. The intermediate language is restricted to single address instructions therefore no use can be made of any double address instructions available on most large computers (and also on some minicomputers).

5.1 Intermediate Language

The intermediate language was influenced by the XPL language and by common minicomputer instruction sets. Most minicomputers use only fixed data types (full word integers) therefore MINCOM should express all character and bit operations in terms of integer operations. Only those operations used by XPL were considered for the intermediate

OPERATION	SEMANTICS
1. Add	OP1 ← OP1 + OP2
2. Subtract	OP1 ← OP1 - OP2
3. And	OP1 ← OP1 & OP2
4. Or	OP1 ← OP1 | OP2

COMPARISON	SEMANTICS
5. OP1 = OP2	OP1=-1 if true else OP1=0
6. OP1 < OP2	OP1=-1 if true else OP1=0
7. OP1 > OP2	OP1=-1 if true else OP1=0
8. OP1 ¬= OP2	OP1=-1 if true else OP1=0
9. OP1 ¬< OP2	OP1=-1 if true else OP1=0
10. OP1 ¬> OP2	OP1=-1 if true else OP1=0

MONADIC OPERATIONS	SEMANTICS
11. Not OP1	OP1 ← -OP1-1
12. Negate OP1	OP1 ← -OP1

DATA MOVEMENT	SEMANTICS
13. Load register	OP1 ← Memory
14. Store register	Memory ← OP1
15. Move register	OP1 ← OP2

CONTROL INSTRUCTIONS

16. Branch to OP2 if OP1 = 0
17. Branch to OP2 if OP1 < 0
18. Branch to OP2 if OP1 > 0
19. Branch to OP2 if OP1 ¬< 0
20. Branch to OP2 if OP1 ¬> 0
21. Branch to OP2 if OP1 ¬= 0
22. Branch unconditionally to OP2
23. Branch to subroutine OP2

24. Enter Macro
25. Exit Macro

DATA DEFINITION

26. Character data
27. Fixed data
28. Label defination
29. Address constant
30. Assign block of storage

PSEUDO-OPERATIONS

31. Set location counter

Figure 5.1 Intermediate Language Definition

language.

The operations included in the intermediate language as described in [9] are given in Figure 5.1.

The XPL input/output functions INPUT(N), OUTPUT(N), and FILE(N,M) were implemented with branches to the assembly language subroutine MONITOR. The only non-character operations used in XPL but not included in the intermediate language were multiply, divide, mod (remainder), shift right and shift left. The Data General Nova computer that was used to test the TWS did not have the hardware multiply or divide option. It was felt that even if a minicomputer had a hardware multiply or divide, a call to the monitor for multiplication and division would add very little extra overhead since special registers or memory locations would have to be loaded in any case.

The shift right and shift left operations could have been included in the intermediate language. They were left out of the intermediate language (and were implemented as monitor calls) for two reasons. For the Nova, the macros were five instructions long whereas the monitor call included only three instructions. Including SHR and SHL as monitor calls meant that all language defined functions could be treated as monitor calls within MINCOM.

The character functions SUBSTR(S,N,M), || (catenate) and BYTE(S,N) would have required very lengthy macros had they been included in the intermediate language. The only other functions required by XPL and included in the monitor were string comparison, string assignment, and integer to character string conversion.

The intermediate language as described above has not defined the operands. The operands are defined within MINCOM. These

definitions may be changed by changing constants within MINCOM. The types of operands that may be specified are given in Figure 5.2.

	OP1	OP2
0.	Register	Register
1.	Register	Memory
2.	Register Register	Register or Memory
3.	Register	undefined
4.	not used	
5.	undefined	Register
6.	undefined	Memory
7.	undefined	undefined

Figure 5.2 Intermediate Language Operand Types

MINCOM assumes that results are left in OP1 and that OP2 (if it is a register) is free after an operation. MINCOM does all the address calculations and indicates indirect addressing for Memory Reference instructions if indirect addressing is required.

5.2 Structure of Programs Produced by MINCOM

XPL variables are known only within their own scope and within higher scopes. Level zero has been assigned to the global scope, level one to a scope nested within level zero, level two to a scope nested within level one and so on. Only some of the XPL phrases indicate the start of a new level. The following is a list of phrases which, when recognized by the parser, cause the level number to be incremented. The corresponding scope ending is also listed.

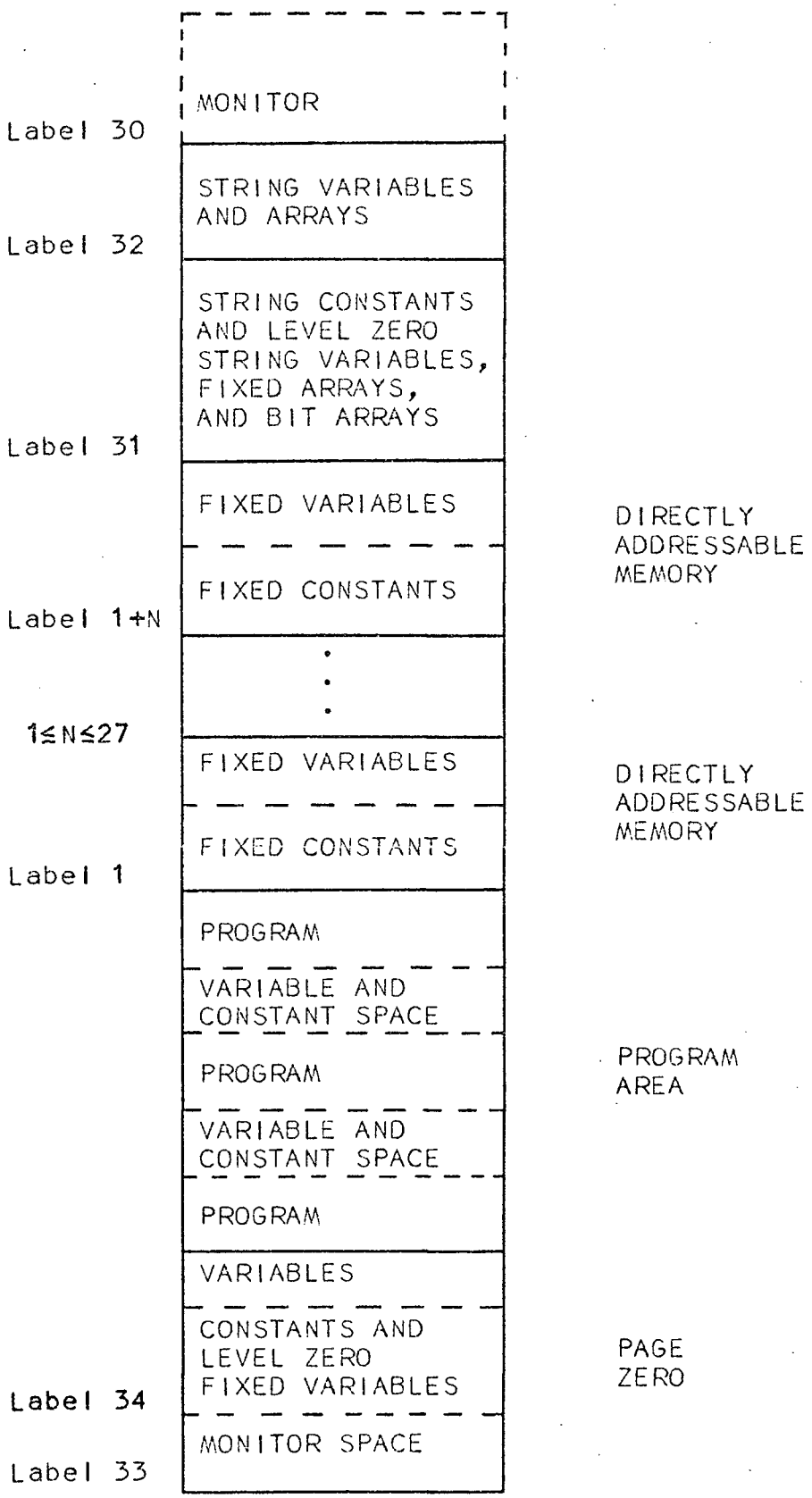


Figure 5.3 Structure of Intermediate Language Programs

Level incremented After	Decremented After
<procedure name>	<ending>
<if clause>	<if statement>
<group head>	<ending>

All labels are defined on level zero and can be used at any level. Procedures must be defined before they can be called.

An intermediate language program is structured as in Figure 5.3. A small area of memory that can be addressed from anywhere within the program must be available for passing parameters to the monitor. Pages of globally addressable memory are defined in MINCOM. This memory is used for fixed constants, address constants, user defined fixed variables and machine defined temporary variables. Constants and variables are assigned space within the program area when MINCOM has used all the available directly addressable memory. Variables defined within the program area might be directly or indirectly addressed depending on their distance from the variable reference.

As an example of the above method of memory assignment consider the Nova. Most of page zero may be used as directly addressable memory. Registers 2 and 3 may be loaded with pointers to two other pages therefore almost 256 x 3 = 768 words of globally addressable memory is available. The sample parser produced for this thesis occupied approximately three thousand words of memory but used less than 200 words of fixed variables, constants and address constants.

All string constants and level zero string variables, fixed arrays and bit arrays are assigned to the area beginning with label 31. Non level zero string variables, fixed arrays and bit arrays are assigned to the area beginning with label 32. All variables within areas 31 and 32 are referenced indirectly and require an address constant in

one of the addressable areas.

Area 32 is reassigned as the level changes, therefore variables assigned in area 32 cannot be initialized. Only level zero variables may be given initial values.

5.3 MINCOM

The XPL to intermediate language translator was modelled after XCOM, the XPL to IBM 360 machine code compiler of the XPL TWS. The XPL grammar was modified as previously described then was input to the grammar analyzer of the XPL system. The scanner routine in the skeleton parser was modified to recognize strings, XPL comments and XPL literals. Only the semantic procedure SYNTHESIZE was completely re-written. The tables produced by the XPL analyzer were combined with the new skeleton parser and the new SYNTHESIZE routine to form an XPL source program. This program was compiled using XCOM to produce an object code MINCOM. The structure of the source program for MINCOM is shown in Figure 5.4.

5.4 Constants within MINCOM

MINCOM was written so that only a very simple macro generator would be required to translate the intermediate language code into a particular minicomputer assembly language. To accomplish that goal, the minicomputer must be described within MINCOM. That is done by changing constants within MINCOM. The constants used for the Nova minicomputer are shown in Figure 5.5. The intermediate language includes thirty-one instructions. Those instructions cannot be changed but the types of operands can. The operands may be registers or memory as specified in Section 5.1. The MINCOM variable INST-TYPE contains the

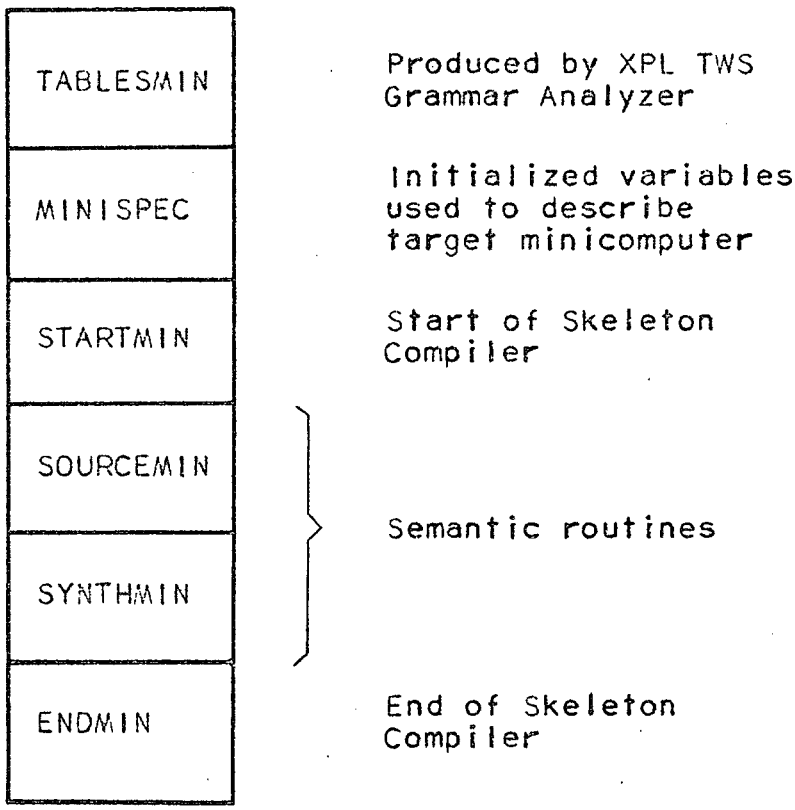


Figure 5.4 MINCOM Source Structure

operand type information (see Fig 5.2) and may be changed by replacing the initial list. The array INST-LENGTH contains the length of each type of instruction in minicomputer words. MINCOM must know what length macro will be generated for each instruction type in order to determine when indirect addressing is required.

The numbers of the scratch registers are stored in the array SCRATCH-REG#. Up to sixteen scratch registers may be specified. MINCOM passes parameters to the monitor via available registers and page zero memory locations. The MON-R-STACK array contains the identification numbers of the registers which can be used to pass parameters to the monitor. The content of register zero was saved if necessary but any other registers were assumed free. Register 2 was used in the Nova implementation when programming examples showed that only about two

```
   /* BEGIN FILE 'MINISPECS'  */

/*  ALL ARRAYS BEGIN WITH INDEX ZERO  */

DECLARE WORD_LENGTH LITERALLY '16';
   /* BIT WORD LENGTH OF THE MINICOMPUTER */

DECLARE PAGE_SIZE LITERALLY '256' ;
   /* PAGE SIZE OF THE MINICOMPUTER */

DECLARE DIR_ADDRESS(2) FIXED INITIAL(32,255,0);
   /* DIRECTLY ADDRESSABLE SPACE.  THE FIRST TWO NUMBERS INDICATE START AND STOP
   ADDRESSES OF FIRST BLOCK.  THE FOLLOWING NUMBER INDICATES A BLOCK OF DIRECTLY
   ADDRESSABLE MEMORY.  THE FIRST ADDRESS IS ASSIGNED LABEL L33.  ANY ADDITIONAL
   PAGES ARE LABELLED WITH A LABEL GREATER THAN  L33.  */

DECLARE NO_OF_INDEX_REG LITERALLY '1';
   /* NUMBER OF INDEX REGISTERS.  THE COMPILER ASSUMES THAT EACH INDEX
   REGISTER WILL BE LOADED WITH A POINTER TO A PAGE OF MEMORY.  PAGES ARE
   LABELLED L1,L2, ... ,L29.  */

DECLARE NO_OF_SCRATCH_REG LITERALLY '2' ;
   /* NUMBER SCRATCH REGISTERS */

DECLARE NO_OF_PAR_REG LITERALLY '1' ;
   /* NUMBER OF REGISTERS USED FOR PASSING PARAMETERS TO THE MONITOR */
   /* (NUMBER OF REGISTERS IN ADDITION TO THE FIRST SCRATCH REGISTER ) */

DECLARE INDEX_REG_STACK(0) FIXED INITIAL(3);
   /* THE ACTUAL INDEX REGISTER NUMBERS */

DECLARE SCRATCH_REG(1) FIXED INITIAL(0,1);
   /* USED TO STORE THE FREE SCRATCH REGISTERS */

DECLARE SCRATCH_REG#(1) FIXED INITIAL(0,1);
   /* USED AS A PERMANENT RECORD OF THE ACTUAL SCRATCH REGISTER NUMBERS */

DECLARE MON_R_STACK(0) FIXED INITIAL(2);
   /* A LIST OF REGISTERS THAT MAY BE USED FOR PASSING PARAMETERS TO THE
      MONITOR */

DECLARE PREVIOUS_PAGE_BOUND FIXED INITIAL(128);
   /* ACTUAL ADDRESS OF LOWER PAGE BOUNDARY */

DECLARE NEXT_PAGE_BOUND FIXED INITIAL(384);
   /* ACTUAL ADDRESS OF NEXT PAGE BOUNDARY */

DECLARE NEXT_INST_COUNTER FIXED INITIAL (256);
   /* STARTING ADDRESS */

DECLARE PAGING_INCREMENT LITERALLY '1' ;
   /* DETERMINES THE NUMBER OF INSTRUCTIONS TO BE ADDED BEFORE THE PAGE
      BOUNDARIES ARE MOVED */

   DECLARE INST_LENGTH(31) BIT(8) INITIAL
      (0,1,1,1,4,5,5,5,5,5,5,1,1,1,1,1,2,2,2,2,2,2,1,1,7,6,2,1,0,1,0,0);

   DECLARE INST_TYPE(31) BIT(8) INITIAL
      (0,0,0,0,0,0,0,0,0,0,0,3,3,1,1,0,3,3,3,3,3,3,7,7,7,7,7,7,7,7,7);

DECLARE LEVEL_ONE LITERALLY '200' ;
   /* STARTING LOCATION FOR NON-LEVEL ZERO SYMBOL TABLE */

DECLARE STS LITERALLY '400';
   /* SYMBOL TABLE SIZE */

DECLARE FCS_SIZE LITERALLY '500';
   /* SPACE IN THE COMPILER FOR THE FIXED CONSTANTS AND ADDRESS CONSTANTS */

DECLARE DATA_PAGES3 LITERALLY '60';
   /* DETERMINES SIZE OF ARRAY USED TO KEEP TRACK OF MINICOMPUTER
         STORAGE SPACE */

DECLARE FCLIM LITERALLY '100';
   /* SIZE OF THE FIX-UP STACK */

DECLARE NO_OF_PROCEDURES LITERALLY '50' ;
   /* NUMBERS OF PROCEDURES ALLOWED */

STRING_LENGTH: PROCEDURE(LEN);
   /* USED TO CALCULATE THE NUMBER OF WORDS USED TO STORE A CHARACTER STRING
   OF LENGTH LEN  CHARACTERS ( LEN=1,2, ...)   */
   DECLARE LEN FIXED;
   RETURN (LEN+1)/2+1;
END STRING_LENGTH;

   /* END FILE 'MINISPECS'  */
```

Figure 5.5 Constants Used to Describe Minicomputer

hundred locations were required for constants and fixed variables. For computers with more than one register available for passing parameters to the monitor, the method used is quite efficient but computers with a single accumulator require two instructions per parameter. A simple change to MINCOM can be made to pass parameters via address constants following the monitor call. MINCOM was not written to pass parameters as address constants because XPL allows expressions to be used as parameters. The value of an expression is normally left in a register. If address constants were used to pass parameters, the register would have to be stored.

5.5 Addressing

Variables and labels may be directly addressed, indirectly addressed, or addressed via index registers. MINCOM was written for computers capable of indirect addressing. A change to the character string and array addressing routine would have to be made if only index register plus displacement type addressing was available.

All constants and variables stored in the predefined areas are directly addressed. All fixed and bit arrays and all character strings are indirectly addressed. Labels are directly addressed only if they are on the current page. MINCOM uses the variables PREVIOUS-PAGE-BOUND and NEXT-PAGE-BOUND as the limits for the current page. These variables must be initialized to the proper memory locations for the first program page. Relocatable programs may be obtained by changing absolute locations to labels with the macro generator. The variable NEXT-INSTRUCTION-COUNTER must be initialized to the first program memory location. Each instruction that is added to the program

increases the NEXT-INSTRUCTION-COUNTER by the length of the instruction.

The page boundaries are moved when the PAGE-POSITION-COUNTER becomes equal to or greater than the PAGING-INCREMENT. The page boundaries are moved by an integer number of paging increments. A floating page such as that employed in the Nova was specified by setting the PAGING-INCREMENT equal to one. The fixed page used in the PDP/8 may be specified by setting the PAGING-INCREMENT to the fixed page size (128_{10}).

5.6 MINCOM Data Structures

McKeeman et al. [3] indicated that for large programs, the largest single cost of translation using their compiler is the cost of symbol look-up. They used a stack for their symbol table. As new variables were declared on a level, they were added to the top of the stack. Symbol look-up was accomplished by starting at the top of the stack and searching down to level zero. This technique allowed variables with the same name but declared on different levels to be distinguished. Any reference to a variable was assumed to be a reference to the variable on the closest level.

In order to reduce the time spent in symbol look-up, MINCOM uses a binary tree to store the symbols and related information. The average length of a binary tree search is approximately $\log_2(n)$ where n is the number of symbols in the tree. The average length of a linear search is n/2. Assuming that two comparisons are required for each step of the binary search, the average binary search requires fewer comparisons when $n > 16$. There will normally be more than sixteen symbols in the symbol table since all machine generated labels are entered. A

single binary tree was not used because symbols must be removed from the symbol table as the level is decreased. A series of stacked binary trees were used for the symbol table in MINCOM. Since the higher levels contain fewer than sixteen symbols, a clear gain in table search time cannot be shown. Experience with XPL programs has shown that many global variables are used therefore the level zero tree is normally much larger than sixteen so that an overall gain is realized. In order to be able to add labels to the level zero tree from any level, a separate data area was assigned to level zero.

5.7 Code Generation

A two hundred and forty byte buffer is kept in memory for the program instructions. A similar buffer is kept for the string constants and level zero arrays. Since each intermediate language instruction occupies six bytes, only forty instructions are stored in memory. The XPL function FILE(N,M) is used to write into and read from a file. N specifies the file number and M the block number. The UBC XPL implementation uses a block size of 240 bytes hence the small buffer. Since MINCOM is a single pass compiler fix-ups are used for every forward branching instruction. All forward branches are assumed to be directly addressable and are coded that way. Each time a reference to an undefined label (both user defined labels and machine defined labels) is made, the instruction location and label number are entered in a fix-up stack. Each time the next instruction counter is incremented, its value is compared with the first entry in the fix-up stack. If the fix-up entry cannot be directly addressed, an address constant is defined and the original instruction is modified.

Because only forty instructions are kept in memory, fix-ups often in--volve file reading and writing. The execution time of MINCOM was not found to be excessive therefore a more elaborate file buffering scheme was not written. When a lable is defined, all references to that label are removed from the fix-up stack.

5.8 Constants

Fixed constants and address constants (label number plus displacement) are kept in memory and are not written into the object code file until after compilation is complete. The constants are stored in a binary tree. As long as addressable space remains for the constants (space in the minicomputer that the intermediate code is ultimately meant for) no constants are repeated. When constants must be inserted within a program, they might be duplicated.

Level zero variables may be initialized with an initial list. If level zero variables are not initialized, then they are set equal to zero. All other variables may not be initialized by the programmer and are not initialized by MINCOM.

6. ANALYZERS

There are two conflicting requirements that a grammar type must meet for minicomputer applications. The first is that the restrictions on the BNF grammar must not be too severe. It should be easy to create and debug BNF grammars acceptable to the grammar analyzer. The second requirement concerns the size of the parser. The purpose of this thesis was to develop a TWS for minicomputers with a minimum memory size of eight thousand words. Half of the available memory must be used for the semantic routines therefore the parser must occupy less than four thousand words. Since an MSP(2,1;1,1) analyzer was available on the IBM 360 at UBC, the first analyzer produced for this thesis was a modified version of the XPL TWS analyzer.

6.1 A Local Context Simple Precedence Grammar

The XPL TWS analyzer was modified to accept only simple precedence grammars (those which require only the top symbol on the parse stack and the next symbol from the input text to make the reduce, stack, or error decision). A simple precedence parser is smaller than an MSP parser and uses smaller tables but simple precedence grammars are difficult to write. In order to extend the class of grammars acceptable to the analyzer, a further modification was made. The parser was written so that it would check input symbols for special keywords. When a keyword was found, a simple precedence grammar associated with that keyword was used to analyze the following statements. The grammar definition input to the analyzer was segmented with $EOG cards. Each grammar was analyzed separately. The tables were combined after an end of file.

The local grammar approach extended the class of acceptable grammars and provided smaller tables than a global simple precedence grammar did. Many small, locally simple precedence grammars might not be globally MSP(2,1;1,1). The simple precedence grammars use a double subscripted array for storing the possible combinations of symbol pairs. A translator's grammar consists of many unrelated statement types. These statements contain a large number of reserved words. Because of the independence of statement types, the n by m (n terminal symbols, m terminal plus non-terminal symbols) array will be mainly zeros indicating illegal combinations. Independent grammars use many small two dimensional arrays of total memory size much smaller than an n by m array.

The grammar in Appendix I was written to correspond to the B.C. Telephone system builder language. The grammar was not written to correspond to any particular class of grammars. After it was segmented by keywords, it was found to be not locally simple precedent. It was also not MSP(2,1;1,1). Rather than force the user of the minicomputer TWS to write complicated grammars, an LR(1) grammar analyzer and parser were produced. The LR(1) grammars use the complete past history of the parse and do not require any special techniques to make use of prefix verbs.

6.2 LR(1) Grammars

Before the LR(1) parsing algorithm produced for this thesis can be described, a few definitions must be given. The parser is a finite state machine (FSM). At each step of the parse, the parser is in one of n finite states. Information associated with that state is used to determine the next state. A configuration with a normal

successor symbol is called a transition. The successor may be the null symbol indicating the end of a production. A configuration with a null successor is called a reduction. States with two or more reductions and states with both transitions and reductions are called inadequate states. They are called inadequate states because one or more symbols from the input text must be investigated before a decision to stack or reduce can be made. The LR(1) grammars considered in this thesis require only one symbol from the input text to make all the stack, reduce or error decisions.

DeRemer [6] used look-ahead sets with each configuration in an inadequate state. The parser compared the next symbol from the input text with each symbol in the look-ahead set associated with a configuration. If a match was found, then the configuration was used to determine the next state. The look-ahead set associated with a transition is the successor symbol. The set associated with a reduction consists of all terminal symbols that may follow the reduction.

A Very Simple LR(1) (VSLR(1)) parser and grammar analyzer were produced for this thesis. A grammar is VSLR(1) if it is LR(1) and each state has zero or one reduction. For LR(1) grammars, the decision between stacking or reducing can always be made without using the look-ahead sets associated with the reductions. If the next input symbol does not match any of the transitions in an inadequate state, then a reduction must be made (or an error has occurred). If the next symbol was in error then that error will be observed in the next non-inadequate state. If the next symbol does match a transition symbol, then that transition must be the correct one because only the next symbol may be used to make the decision. The flowchart for the VSLR(1) parsing

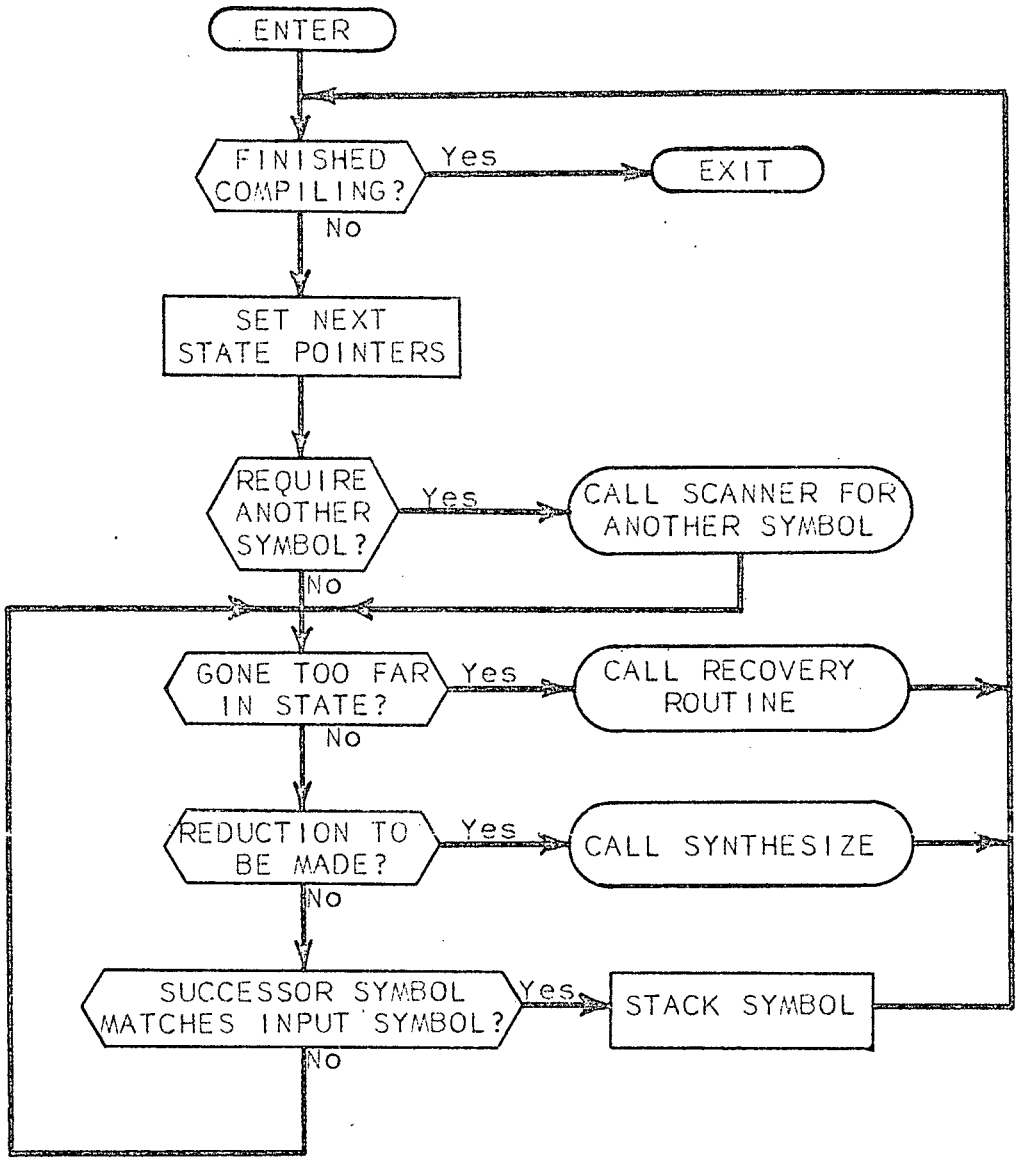


Figure 6.1 VSLR(1) Parsing Algorithm

```
 1   <PROGRAM>  ::=  <STATEMENT LIST> END

 2   <STATEMENT LIST>  ::=  <STATEMENT>
 3                       |  <STATEMENT LIST> <STATEMENT>

 4   <STATEMENT>  ::=  <ID LIST> <ASSIGNMENT> ;
 5                  |  <ID LIST> <OTHER ASSIGNMENT> .
 6                  |  <IDENTIFIER> <ASSIGNMENT> .
 7                  |  <IDENTIFIER> <OTHER ASSIGNMENT> ;

 8   <ASSIGNMENT>  ::=  = <IDENTIFIER>

 9   <OTHER ASSIGNMENT>  ::=  = <IDENTIFIER>

10   <ID LIST>  ::=  <IDENTIFIER> <IDENTIFIER>
11                |  <IDENTIFIER> <ID LIST>
```

Figure 6.2 A Non SLR(1) Grammar

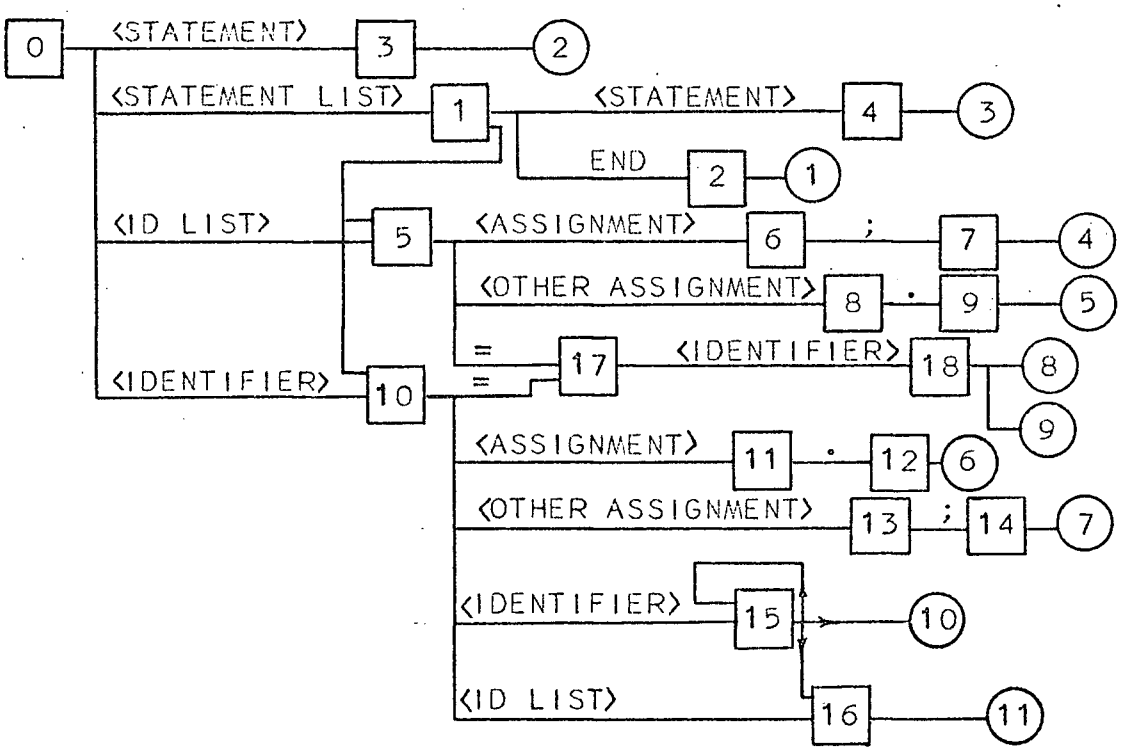


Figure 6.3 FSM For Grammar of Fig. 6.2

algorithm is given in Figure 6.1.

Although the VSLR(1) parser accepts a wide class of grammars, some common LR(1) grammars are rejected. For example, the following grammar is not VSLR(1).

```
1.  <program>        ::= <statement list>  END

2.  <statement list>::= <statement>
3.                    | <statement list>  <statement>

4.  <statement>      ::= <assignment>;
5.                    | <other assignment> .

6.  <assignment>     ::= <identifier> = <identifier>

7.  <other assignment>::= <identifier> =  <identifier>
```

The equal right parts of production 6 and 7 force two reductions in one of the states of the FSM. The next input symbol must be used to determine which reduction to make. If it is a semicolon then production 6 should be used otherwise production 7 should be used. The SLR(1) parser of DeRemer [6] is able to make the decision by storing a semicolon with reduction 6 and a period with reduction 7. As long as the look-ahead sets are disjoint, the SLR(1) parser can proceed.

Another example of a non-VSLR(1) grammar is given in Figure 6.2. The grammar is LR(1) but not SLR(1). The FSM configuration has been given in Figure 6.3. A look-ahead set cannot be used to decide between reductions 8 and 9 in state 18 because the look-ahead sets are dependent on the previous states. The previous states must be used to indicate whether a semicolon indicates that production 8 or 9 should be used in state 18. DeRemer [6] solved this type of problem by splitting states. States 17 and 18 may be repeated as states 19 and 20. State 10 then leads to state 19 and the look-ahead sets in state 20 will be disjoint.

One of the goals of this thesis was to produce a small parser which could use a wide class of grammars. A small routine was added to the VSLR(1) parser in order to extend it to an LR(1) parser. All the information required to calculate look-ahead sets is contained in the state array. It was therefore possible to make the parser calculate look-ahead sets when required. The approach required no state splitting and no storage of look-ahead sets. The algorithm used to calculate the look-ahead sets occupied about 220 words on the Nova. Since that routine was not much larger than the comparison algorithm required to search through stored look-ahead sets, the approach is consistent with the goal of small memory size at, possibly, the expense of execution time. The execution time of the parser look-ahead algorithm (PLALR(1)) whould be longer than that of a stored look-ahead set parser but since both methods involve a search (PLALR(1) searches through the state matrix for a symbol, LALR(1) searches through a look-ahead set for a symbol) the difference should not be too significant.

Another way of looking at the PLALR(1) parsing algorithm is to consider the parser to be in one of two modes. The first mode is a VSLR(1) mode. The parser continues in the VSLR mode until more than one reduction is encountered in a state. A search mode is then entered. In effect, the first reduction is attempted. If no error results after the next input symbol has been trial stacked then the original decision was the correct one. If an error is encountered, the following reduction is attempted. The process continues until a correct reduction is found or until the last reduction in the original state must be attempted. The last reduction must be the correct one and the parser is put back into the VSLR mode.

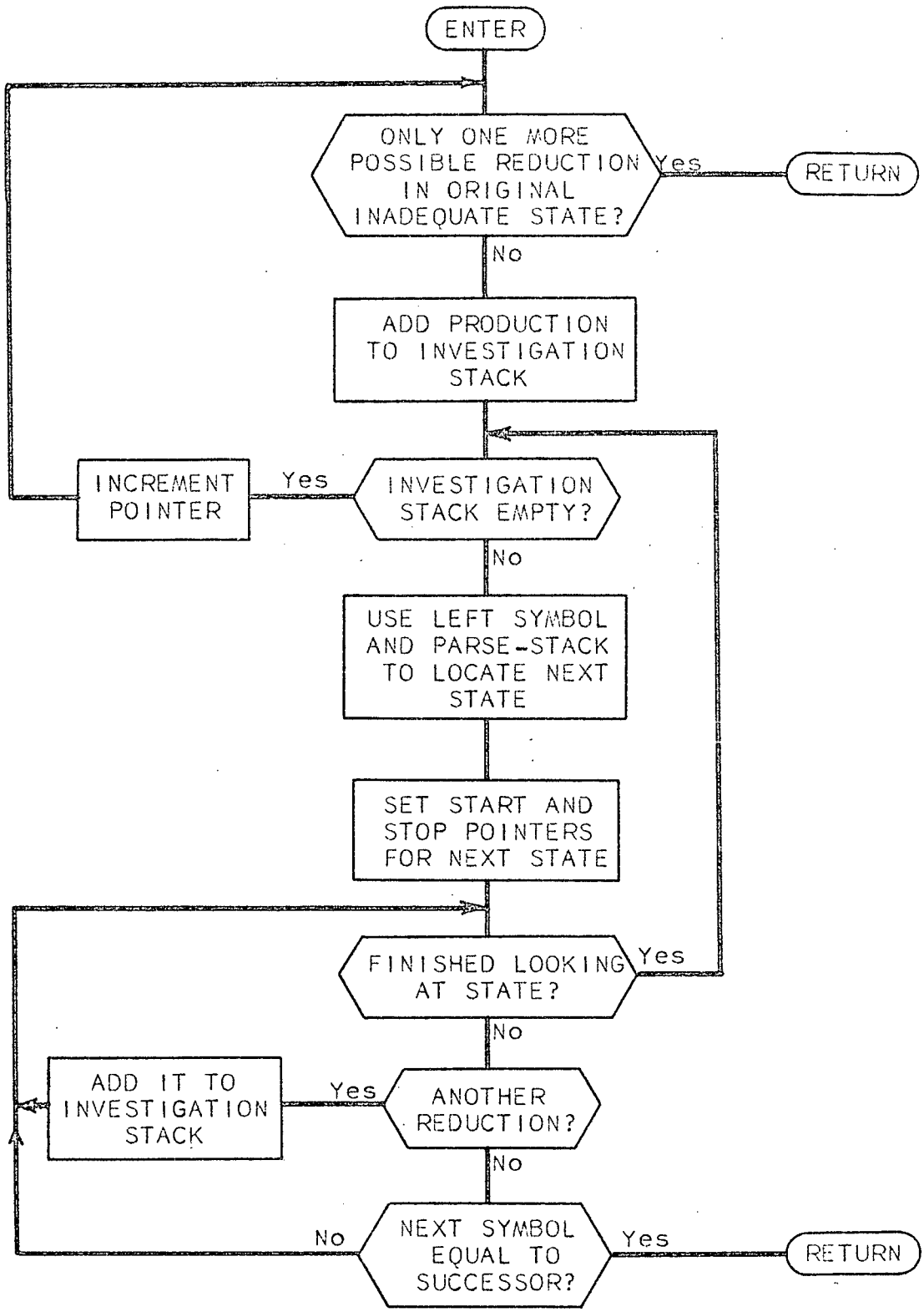


Figure 6.4 PLALR(1) Search Algorithm

The example of a non-SLR(1) grammar was a very simple one. In general, a finite number of reduction decisions must be investigated before a look-ahead set can be determined. The PLALR(1) algorithm makes use of the fact that the parse stack can never increase as nested reductions are encountered. The parser must make a trial reduction, obtain the next possible state, check the allowable terminal symbols and, if a match occurs between one of them and the next input symbol, proceed with the reduction under investigation. If another reduction is encountered before a match is found, then its next state may be determined from the parse stack. The algorithm is relatively simple for the LR(1) case because nested trial reductions do not affect one another. Only the original parse stack need be used to determine next states. A history of trial reductions is not kept.

The flow chart for the search algorithm appears in Figure 6.4. The algorithm may be generalized to work for LR(k) grammars, although all inadequate states would have to be investigated, rather than only those with more than one reduction. The search would terminate after k symbols had been matched. The values of k for an inadequate state would have to be stored in the state array. If the previous parse stack is stored before each search and comparison of an input symbol, then the algorithm may proceed as in the LR(1) case. A mismatch at any stage must restore the previous parse stack and input symbol. It is necessary to maintain parse stacks in the general case because matched symbols must be stacked before the algorithm can proceed to the next symbol.

6.3 LR(1) Analyzer Algorithm

The LR(1) analyzer program accepts a grammar written in a modified BNF and produces all the tables required by the PLALR(1) parser described in the previous section. The tables are output as XPL declaration statements. An example of the tables produced by the grammar analyzer is shown in Figure 6.5. The tables were produced for the grammar of Figure 6.2.

```
--- CARD OUTPUT ---I   DECLARE V CHARACTER(18) INITIAL ( ',', ';', '=', 'END', '<IDENTIFIER>');
--- CARD OUTPUT ---I   DECLARE MAX_RL LITERALLY'12';
--- CARD OUTPUT ---I   DECLARE CHAR_INDEX(13) BIT(3) INITIAL (0, 1, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5,
--- CARD OUTPUT ---I      6);
--- CARD OUTPUT ---I   DECLARE PR_LENGTH(11) BIT(3) INITIAL(0, 3, 1, 2, 3, 3, 3, 3, 2, 2, 2, 2);
--- CARD OUTPUT ---I   DECLARE LEFT_PART(11) BIT(4) INITIAL (0, 6, 10, 10, 8, 8, 8, 8, 9, 11, 7, 7);
--- CARD OUTPUT ---I   DECLARE STATE (45) BIT(5) INITIAL ( 10, 1, 3, 5, 2, 4, 5, 10, 0,1, 0,2, 0,3,
--- CARD OUTPUT ---I      17, 6, 8, 7, 0,4, 9, 0,5, 17, 11, 13, 15, 16, 12, 0,6, 14, 0,7, 15, 16,
--- CARD OUTPUT ---I      0,10, 0,11, 18, 0,8, 0,9, 0);
--- CARD OUTPUT ---I   DECLARE STATE_P(19) BIT(6) INITIAL ( 0, 4, 8, 10, 12, 14, 17, 18, 20, 21, 23,
--- CARD OUTPUT ---I      28, 29, 31, 32, 34, 38, 40, 41, 45);
--- CARD OUTPUT ---I   DECLARE STATE_SYM(18) BIT(4) INITIAL (0,  10, 4, 8, 8, 7, 9, 2, 11, 1, 5, 9,
--- CARD OUTPUT ---I      1, 11, 2, 5, 7, 3, 5);
PUNCHING COMPLETE,
EXECUTION TERMINATED
```

Figure 6.5 Example of Tables Produced by Grammar Analyzer

The tables, parser and semantic routines may be compiled by XCOM to produce IBM 360 object code or by MINCOM to produce an intermediate language program. The analyzer has been written so that a flag may be set to indicate which compiler will be used to translate the tables. The only difference in the tables is the vocabulary declaration. XCOM accepts an n-dimensional array of symbols of unspecified length. MINCOM does also but assigns a default length to each character variable. Since the vocabulary will never be altered, it should be packed as efficiently as possible. The single character symbols should only occupy one byte rather than the default length of eighty bytes. The vocabulary meant for MINCOM is output as a single character variable

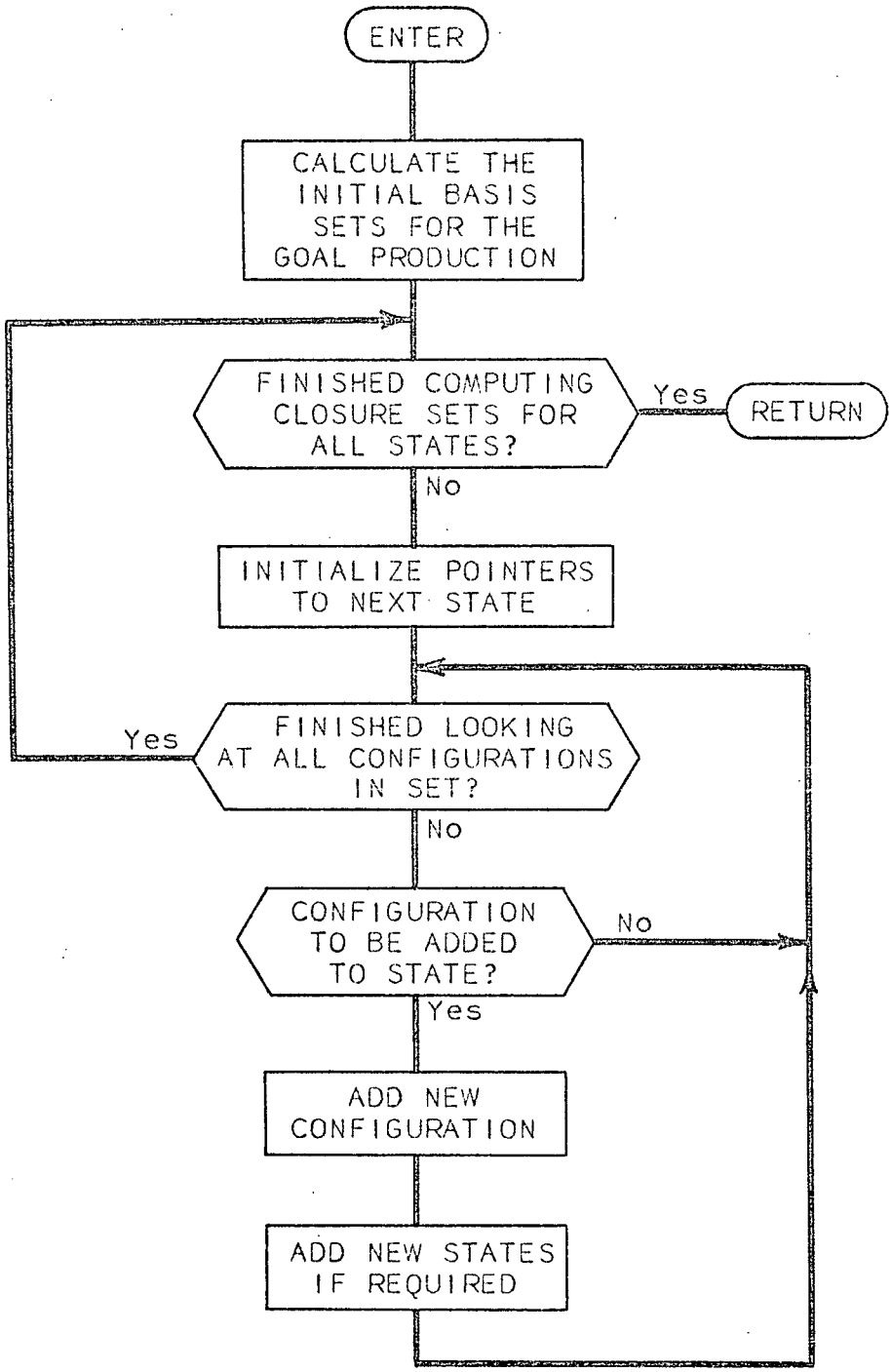


Figure 6.6 LR(1) Analyzer Algorithm

of length equal to the total number of characters in the vocabulary. The array CHAR-INDEX is used to store pointers to the individual symbols.

6.4 Other Data in Output Declarations

The literal MAX-RL gives the length of the longest word in the vocabulary. That information is used by the scanner to quickly decide whether or not a symbol can be a reserved word. The arrays PR-LENGTH and LEFT-PART provide all the information required about the productions in the grammar. PR-LENGTH gives the number of symbols in the right part of the productions. LEFT-PART gives the symbol numbers associated with the left-part non-terminal symbols.

The main analyzer routine is used to calculate the remaining three arrays. The array STATE contains all successor information. STATE-P is used to store pointers into the state array. STATE-SYM contains the symbol number used as an entry to a state.

6.5 State Algorithm

Each state consists of a set of configurations composed of a basis set unioned with a closure set. Each transition configuration in a state forms a basis configuration in a following state. The successor symbol of a basis configuration is the symbol that follows the previous transition successor symbol in the production. The closure set of a state consists of all productions with a left part equal to a successor of the state. The successor symbol of a closure configuration is the first symbol in the right part of the production.

The flowchart for the routine used to calculate the state array is given in Figure 6.6. It is similar to the algorithm outlined

by DeRemer [6]. The routine begins in state zero with the first production called the goal production. State zero is initialized to the single configuration consisting of the goal production. The first symbol of the right part of the goal production is expanded by creating new states with all the symbols in the right part of the goal production used as successors. For example, consider the grammar of Figure 6.2. State zero is initialized to the goal production:

<program> ::= <statement list> END

with the successor '<statement list>' leading to state one. State one now consists of the goal production with 'END' a successor leading to state two. Since there are no more symbols in the goal production, state two calls for a reduction (see Figure 6.3). The algorithm continues by expanding the non-terminal symbols in all the states.

One or more closure configurations must be added to a state for each non-terminal successor in a state. If a closure configuration has a successor already existing in the state then the new configuration (with the successor pointer moved by one position) must be added to the basis set of the indicated successor state. All productions are expanded completely once. Productions may be expanded more than once only if two or more configurations in a state have the same successor symbol. This approach produces a minimal finite state machine. All non-terminal successors in a state produce additional closure configurations. The routine adds closure configurations to all states beginning with state zero and finishing with the last state. Any state that has already been analyzed for closures but whose basis set is augmented, is re-analyzed.

After all basis and closure configurations have been calculated,

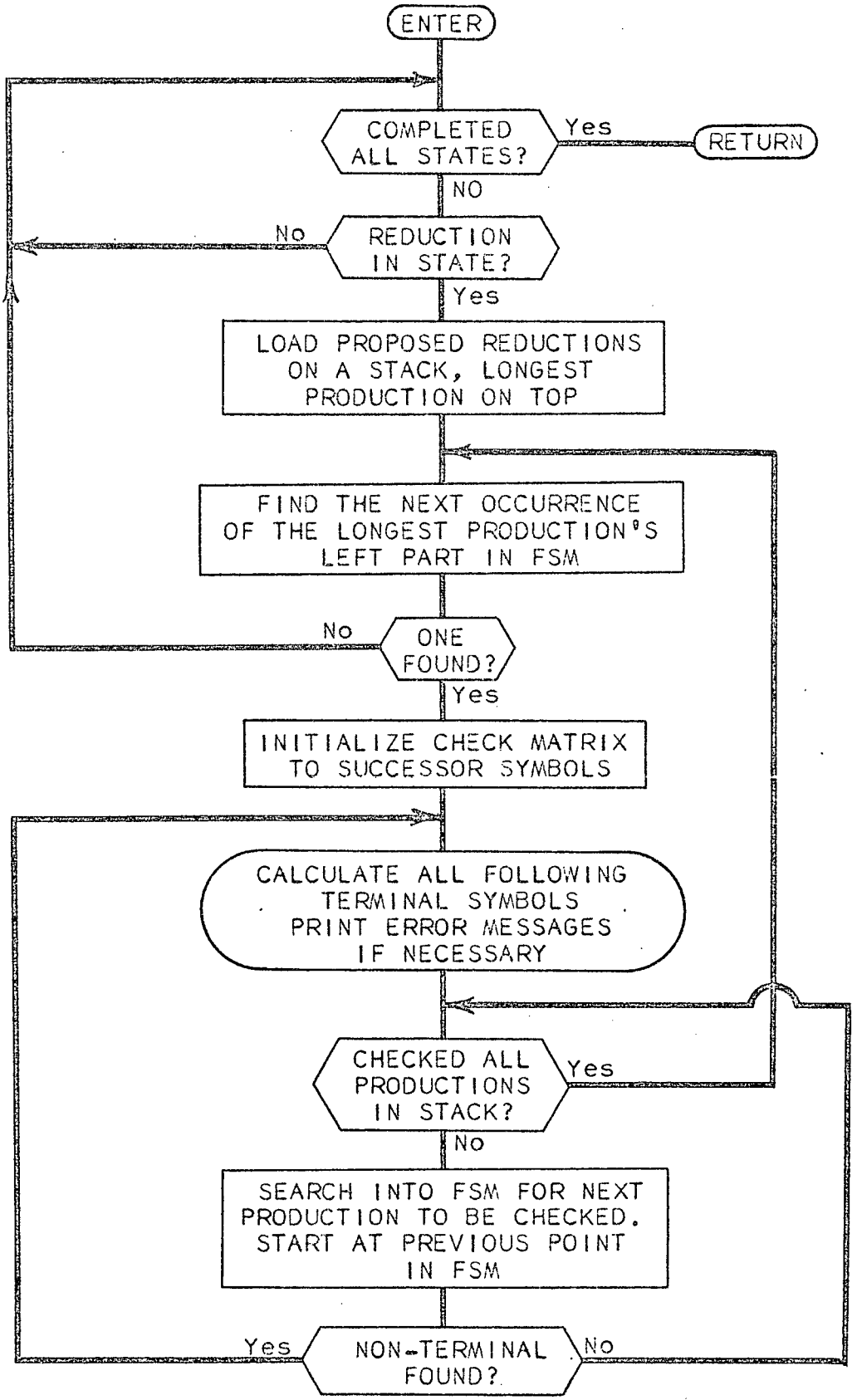
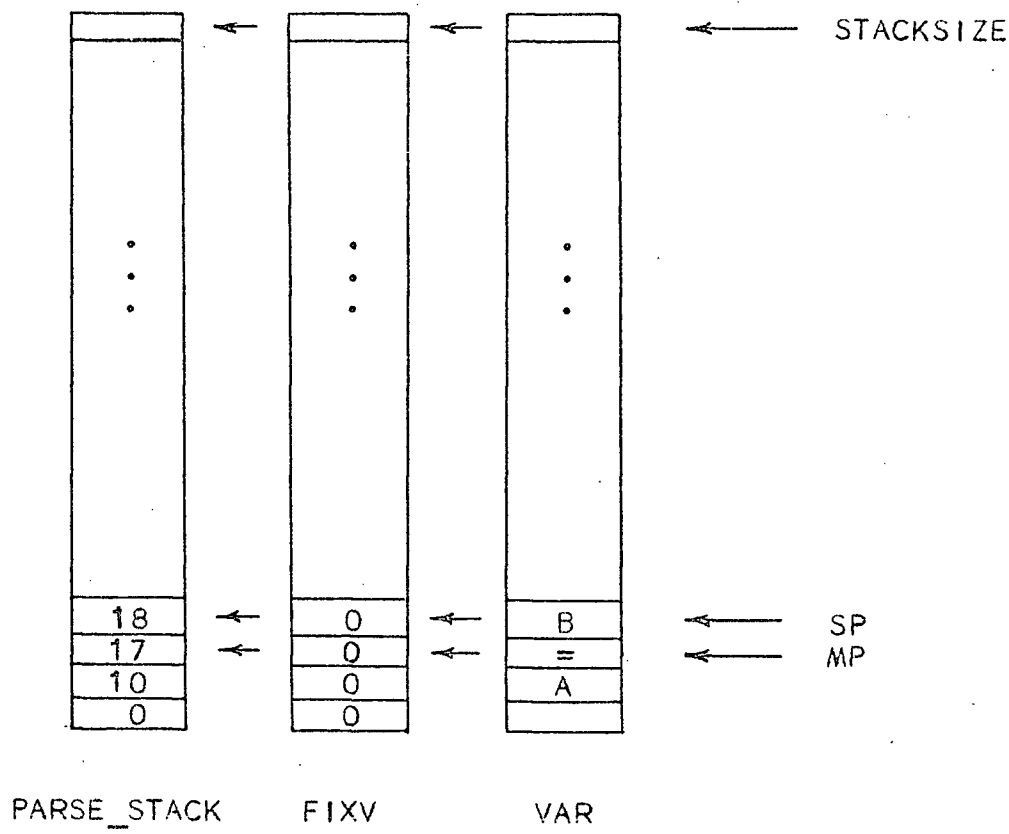


Figure 6.7 Error Detection For LR(1) Analyzer



The contents of the parse stacks indicate values associated with the example of Figure 7.1 after the first line has been entered.

Figure 6.8 Skeleton Parser Data Structure

the finite state machine is checked for ambiguities (non-LR(1) grammars). Next all redundent information is removed from the state array, the reductions are moved to the end of each state and the tables are produced.

6.6 Detection of Non-LR(1) Grammars

A grammar is not LR(1) if more than one symbol from the input text is required to make a stacking or reducing decision. The algorithm operates as in Figure 6.7. It calculates all the look-ahead sets associated with each inadequate state. Only the look-ahead sets that may occur with the same left context are compared within an inadequate state. A grammar is LR(1) if those sets are disjoint.

6.7 Skeleton Parser Variables

All the parsers produced for this thesis interface with the semantic routines SYNTHESIZE in the same manner as the XPL skeleton compiler of McKeeman et al. [3]. Three parallel stacks are used by the parsing algorithm. These stacks are shown of Figure 6.8. The PARSE-STACK contains information indicating the past history of the parse. The character array VAR contains the actual symbols input to the parser and the array FIXV contains the integer value of any number input to the parser. The semantic routine SYNTHESIZE is called when a reduction has been recognized but has not yet been done. The parameter passed to SYNTHESIZE indicates the production number of the proposed reduction. The global variables MP, MPP1, and SP point into the parse stacks. MP points to the left symbol of the recognized productions, SP points to the right symbol and MPP1 is equal to MP+1.

7. ERROR RECOVERY FOR THE PLALR(1) PARSER

The parser was meant for use with minicomputers of limited memory size. Elaborate error correction techniques such as described by Levy [10] could not be used because of the memory size limitation. McKeeman et al. [3] used a simple method of error recovery. Their parser, when it discovered an error, called a recovery routine. The recovery routine scanned the input text for special single characters. Once a special character was found, the input stack was reduced until a legal symbol pair was found. An input symbol was discarded on a second call to the routine. The method worked well as long as no significant reserved words (such as PROCEDURE in XPL) were ignored.

The first error recovery routine used with the PLALR(1) parser was very similar to McKeeman's. It attempted to make use of the possibility that the translator was being used in an interactive mode. If an error occurred, the parser was backed up to the last symbol used in a reduction, the remainder of the current input line was deleted, and a message was printed giving the last symbol used. Error correction could then take place with no error in the output data. If further errors occurred, the parse stack was reduced and input symbols were discarded until a correct reduction was found.

For batch processing of programs, the method outlined did not work well. An error in one line of an input program produced errors throughout a whole section of program. One way to avoid losing important reserved words is to make an attempt at completing the incorrect statement rather than deleting it and searching for a restarting place. A very simple routine was written to do that.

```
A=B.
        8   =   B
        6   A   .
        2   .   .
A=B;
        9   =   B
        7   A   ;
        3   .   ;
A,B,C=D.
        10  B   C
        11  A   =
        9   =   D
        5   =   .
        3   ;   .
A B C=D;
        10  B   C
        11  A   =
        8   =   D
        4   =   ;
        3   .   ;
A=B
C=F.
        9   =   B
*** ERROR, SYNTAX ERROR AT SYMBOL C
        7   A   C
        3   ;   C
        8   =   F
        6   C   .
        3   C   .
A B C.
        10  B   C
        11  A   .
*** ERROR, SYNTAX ERROR AT SYMBOL .
A=B.
        9   .   A
        5   .   .
        3   .   .
A=B;
        9   =   B
        7   A   ;
        3   .   ;
A,B=.
        10  A   B
*** ERROR, SYNTAX ERROR AT SYMBOL .
        9   =   .
        5   =   .
        3   ;   .
C,D,E = F.
        10  D   E
        11  C   =
        9   =   F
        5   =   .
        3   .   .
END
        1     END
#EXECUTION TERMINATED
```

Figure 7.1 Sample program showing Error Recovery

The recovery routine, when first called, inserted the shortest possible correct symbol as the next input symbol. The symbol was inserted only as far as the parser was concerned. No actual change was made to the input text. A second successive call to the recovery routine caused the original input symbol to be discarded. If a symbol was mispelled or missing, and only one symbol was allowed in the context, then error correction resulted. In other cases, error recovery only occurred.

The recovery routine, as described above, was implemented with the grammar of Figure 6.2. The recovery routine did not function well because most possible syntax errors could not be interpreted as single errors. For example, an incorrect statement such as "A = B D." resulted in the recognition of "A = B" as a statement. Attempts at resolving "D." followed by a correct statement failed.

The recovery routine was modified to enter a search mode after inserting a correct symbol. During search mode, a call to the recovery routine caused all symbols stacked since the last reduction to be discarded. Search mode was left after N (N was set to 5) symbols were read from the input text.

An example of the PLALR(1) parser with error recovery implemented on the IBM 360 for the grammar of Figure 6.2 is shown in Figure 7.1. All input begins near the left margin. The semantic routine printed the production number with the left and right symbols of the production. The parser was used in an interactive mode therefore the semantic routine output follows each input line. The first four statements show the four correct types of statements. Notice that all special characters not defined in the grammar may be used as delimiters. The first error was the omission of a period or a semicolon. The parser inserted

a semicolon and recovered from the syntax error (recovered from, not corrected, the error because a period may have been intended). The second error could have been a typing mistake where a space was entered instead of an equal sign. The parser recognized an <id list> before discovering the error. At that point it inserted an equal sign and discovered an illegal period. The period was discarded, the <identifier> A was used as the right part of <other assignment> , =B was discarded, and the period was used to complete the statement. Error recovery took place but one statement was translated incorrectly. The third error was the omission of an <identifier>. The parser inserted an <identifier> and corrected the error. Even though the parser was able to correct the missing <identifier> in the last error, the semantic routine would not find a period in its <identifier> symbol table and a translation error would still result.

An error recovery routine which discards incorrect statements rather than completes them would have recovered from the second error without affecting the following statement. No difficulties arise when a program consists only of a statement list. But when a hierarchy of statement types is used, one discarded statement may prevent many following statements from being translated correctly.

The parser recognizes only syntax errors. Logical errors in a program can never be recognized but other types of errors must be detected. A typical translator must check all information input as a program for such things as size limits on numbers, possible length limits on <identifier>s and declaration of <identifier>s before use elsewhere. These errors must be detected with the semantic routines. There is no simple way of recognizing them within the parser.

8. RESULTS

The constants in MINCOM were set to correspond to the requirements of a Data General Nova minicomputer. A macro generator was written to translate the intermediate language output by MINCOM to Nova assembly language. An assembly language monitor was written for the Nova. Only the teletype input and output, high speed paper tape reader and punch were used by the monitor therefore the FILE(N,M) input and output in the XPL language could not be used in programs meant to run with the Nova monitor.

The VSLR(1) and PLALR(1) parsers were translated into Nova assembly language. The size of the PLALR(1) parser is given in Figure 8.1. The PLALR(1) parser is the VSLR(1) parser with a search procedure added. The search procedure occupied about 220 words of memory.

	16 bit words
Program size	1650
Page zero space used	190
String constants, string variables and parse stacks	600
Monitor	670

Figure 8.1 PLALR(1) Nova Memory Requirement

The XPL grammar in Appendix II, the B.C. Telephone system builder grammar in Appendix I, and a graphics language grammar [11] were input to the XPL MSP analyzer and to the minicomputer LR(1) analyzer The table size for each grammar is given in Figure 8.2. All data was assumed efficiently packed. The MSP table size for the B.C. Telephone system builder grammar is approximate since the grammar was not MSP(2,1; 1,1).

	MSP(2,1;1,1)	PLALR(1)
XPL	2754 bytes	1615 bytes
B.C. Tel system builder	1948 bytes	1224 bytes
Graphics language	2037 bytes	1353 bytes

Figure 8.2 Table sizes for various grammars

The XPL grammar and the B.C. Telephone system builder grammars were found to be VSLR(1). The graphics language was SLR(1).

The tables for the B.C. Telephone system builder were combined with the PLALR(1) parser and translated into Nova assembly language by MINCOM and the macro generator. An example of the resultant program running on the Nova is given in Figure 8.3. The parser was used in an interactive mode. The parser printed a semicolon when input was required. The semantic routine printed the production number with the left and right symbols in the production. The example shows how data statements are analyzed by the parser. The semantic routine that was used with the parser is given below:

```
SYNTHESIZE:  PROCEDURE (PR#);
   DECLARE VAR NESTED;
   DECLARE PR# FIXED;
   OUTPUT='   ' || PR# ||   '  ' ||   VAR(MP) || ' ' || VAR(SP);
END SYNTHESIZE;
```

The example only shows which productions are recognized by the parser. It does not indicate the meaning of the input statements or what a B.C. Telephone system builder actually does. A complete system builder would contain a much larger semantic routine (written in XPL). The XPL CASE statement would be used to execute a set of statements associated with a production number. The system builder semantic routine would output messages to direct the user after appropriate productions had

been recognized. It would analyze the parse stacks and translate the input into a system data base. The example shows that a parser for a complex translator may be produced by the programs of the minicomputer TWS and run on a minicomputer with small memory size.

```
:.DICT
:'TUSK','PAVIL','ALARM 1'
      31 TUSK TUSK
      32 TUSK PAVIL
      32 PAVIL ALARM 1
:'ALARM 2','71F TXA','NORMAL'
      32 ALARM 1 ALARM 2
      32 ALARM 2 71F TXA
      32 71F TXA NORMAL
:.END
      14 NORMAL END
      4 . END
      2 END END
:.MASTER KAMLOOPS,REGISTER-BIT 5
      16 KAMLOOPS
:.VERSION VØ8A
      17 . VØ8A
:.STATION GREEN,Ø,LAST-CONTROL-EQUIPPED-IS 15
      2Ø STATION
      18 .
:.STATION JIM,1,LAST-CONTROL-EQUIPPED-IS 39
      2Ø STATION
      19
:.END
      15   END
      5 . END
      3 END END
:.STATION TUSK
      23 TUSK TUSK
:.GROUP Ø
      24 .
:Ø,A,'71F TXA' <'ALARM 1'> 'NORMAL 71F
      29 A A
      28   A
      31 71F TXA 71F TXA
      31 ALARM 1 ALARM 1
      31 NORMAL 71F NORMAL 71F
:.END
      33 < NORMAL 71F
      27 A .
      25 . .
      21 TUSK END
      6 . END
      3 END END
:END
      1   END
```

Figure 8.3a Sample Run Using Grammar of Appendix I

```
:.LINK 1 IS-TYPE FV14
     58    FV14
:FV14-TXA IS TUSK,5,13,EVALUATE-AFTER-DELAY 2
     63
     61 FV14-TXA
     59
:IF-CXR IS WATT,5,17,EVALUATE-AFTER-DELAY 3
     63
     61 IF-CXR
     60
:IF-TXA IS PAVIL,5,14,EVALUATE-AFTER-DELAY 5
     63
     61 IF-TXA
     60
:.END
     57 FV14 END
     8 . END
     2 END END
:.TCONTROL-SEQUENCE SUTB
     77 SUTB SUTB
:.VERIFICATION 'START-UP OF ALTERNATE TXMTR','TUSK
     31 START-UP OF ALTERNATE TXMTR START-UP OF ALTERNATE TXMTR
      32 START-UP OF ALTERNATE TXMTR TUSK
:RB21
     78 . TUSK
     79 RB21 RB21
:NB21
     80 RB21 NB21
:.END
     76 SUTB END
     10 . END
     3 END END
:END
     1    END
```

Figure 8.3b Sample Run Using Grammar of Appendix I

9. CONCLUSIONS

The purpose of this thesis was to develop a translator writing system (TWS) for minicomputers. The TWS was to consist of programs which were to run on a large computer such as an IBM 360 and produce translators for any specified minicomputer.

In order to make the TWS target computer independent, the high level language XPL was used for the semantic portion of a translator. An XPL to intermediate language compiler was written to aid in the translation of XPL programs into minicomputer assembly language programs.

Three approaches were tried in an attempt to make a wide class of grammars acceptable to the TWS. The first grammar analyzer produced was a modified version of the MSP(2,1;1,1) analyzer of the XPL TWS. It produced tables that were smaller than those produced by the XPL TWS analyzer but the grammars accepted by the analyzer were difficult to write.

The parsing algorithm used by DeRemer [6] for SLR(1) grammars was modified to accept a smaller class of grammars. The VSLR(1) analyzer accepted the B.C. Telephone system builder grammar. A small addition to the parser (requiring no change in the tables produced by the analyzer program) changed it into an LR(1) parser. Grammars for practical translators will correspond in complexity to the B.C. Telephone system builder grammar used in this thesis as an example. Since that grammar was VSLR(1), the LR(1) grammar analyzer and parser produced for this thesis should be adequate for most practical applications.

Further work is required to modify MINCOM and extend the parsing algorithm and analyzer to LR(k) grammars. The version of MINCOM produced for this thesis does not allow memory to memory

instructions in the intermediate language. Since some minicomputers (such as the DEC PDP/11) use memory to memory instructions, a saving in memory space and execution time could be obtained for those computers by extending MINCOM.

A PLALR(k) parser was outlined in the thesis. The idea was not developed further because of time limitations and because LR(1) grammars seemed to be adequate for real-time translators. Very complex grammars could require LR(k) parsers. For these grammars, the look-ahead sets are very large, therefore the parser look-ahead scheme could very well be the only feasible approach for minicomputer applications. An interesting comparison could be made between the execution times of a PLALR(k) parser and a parser using stored look-ahead sets.

APPENDIX I Example Translator Grammar

```
GRAMMAR ANALYSIS ---JM VERSION--- ANALYZER VERSION OF DEC 72

TODAY IS 03-21-73

                P R O D U C T I O N S

$P

  1   <P>  ::=  <STATEMENT LIST> END

  2   <STATEMENT LIST>  ::=  <STATEMENT>
  3                        |  <STATEMENT LIST> <STATEMENT>

  4   <STATEMENT>  ::=  . DICT <DICT END>
  5                  |  . MASTER <MASTER END>
  6                  |  . STATION <STATION END>
  7                  |  <IF STATEMENT LIST>
  8                  |  . LINK <LINK END>
  9                  |  . TCONTROL <TCONTROL END>
 10                  |  . TCONTROL-SEQUENCE <TCONTROL_SEQUENCE END>

 11   <IF STATEMENT LIST>  ::=  <IF START> <IF LIST END>

 12   <IF START>  ::=  LINK-TYPE <IDENTIFIER>
 13                |  COMMAND-TYPE <IDENTIFIER>

 14   <DICT END>  ::=  <PHRASE LIST> . END

 15   <MASTER END>  ::=  <MASTER START> <VERSION> <MASTER LIST> . END

 16   <MASTER START>  ::=  <IDENTIFIER> REGISTER-BIT <NUMBER>

 17   <VERSION>  ::=  . VERSION <IDENTIFIER>

 18   <MASTER LIST>  ::=  . <M STATE>
 19                    |  <MASTER LIST> . <M STATE>

 20   <M STATE>  ::=  STATION <IDENTIFIER> <NUMBER> LAST-CONTROL-EQUIPPED-IS <NUMBER>

 21   <STATION END>  ::=  <STATION> <GROUP> <STATION POINT LIST> . END
 22                    |  <STATION> <LIST> . END

 23   <STATION>  ::=  <IDENTIFIER>

 24   <GROUP>  ::=  . GROUP <NUMBER>

 25   <STATION POINT LIST>  ::=  <POINT>
 26                           |  <STATION POINT LIST> <POINT>

 27   <POINT>  ::=  <POINT START> <PHRASE LIST> <POINT END>

 28   <POINT START>  ::=  <NUMBER> <ALARM>

 29   <ALARM>  ::=  A
 30              |  I
```

```
31   <PHRASE LIST>  ::=  <STRING>
32                    |  <PHRASE LIST> <STRING>

33   <POINT END>  ::=  < <PHRASE LIST> > <PHRASE LIST>

34   <LIST>  ::=  NEVER-DISABLE <NUMBER> <NUMBER>
35              |  <LIST> NEVER-DISABLE <NUMBER> <NUMBER>

36   <IF LIST END>  ::=  <IF LIST> . END

37   <IF LIST>  ::=  <IF STATEMENT>
38                |  <IF LIST> <IF STATEMENT>

39   <IF STATEMENT>  ::=  . IF <EXPRESSION> <VERB> <SPECIAL ID LIST>
40                     |  . IF-NOT <EXPRESSION> <VERB> <SPECIAL ID LIST>
41                     |  SET <IDENTIFIER>
42                     |  SET <IDENTIFIER> CHECK-AFTER-DELAY <NUMBER>

43   <EXPRESSION>  ::=  <EXPRESSION> OR <TERM>
44                   |  <TERM>

45   <TERM>  ::=  <PRIMARY> AND <TERM>
46             |  <PRIMARY>

47   <PRIMARY>  ::=  <IDENTIFIER>
48                |  ( <EXPRESSION> )

49   <VERB>  ::=  THEN-PRINT
50             |  THEN-SERVICE-FAIL-SO-PRINT
51             |  THEN-MINOR-ALARM-SO-PRINT
52             |  THEN-REFUSE-AND-PRINT

53   <SPECIAL ID LIST>  ::=  <SPECIAL ID>
54                        |  <SPECIAL ID LIST> <SPECIAL ID>

55   <SPECIAL ID>  ::=  <STRING>
56                   |  % <STRING>

57   <LINK END>  ::=  <LINK START> <LINK LIST> . END

58   <LINK START>  ::=  <NUMBER> IS-TYPE <IDENTIFIER>

59   <LINK LIST>  ::=  <LINK>
60                  |  <LINK LIST> <LINK>

61   <LINK>  ::=  <IDENTIFIER> IS <IDENTIFIER> <ENDING>

62   <ENDING>  ::=  <NUMBER> <NUMBER>
63               |  <NUMBER> <NUMBER> EVALUATE-AFTER-DELAY <NUMBER>
64               |  <NUMBER> <NUMBER> ABNORMAL
65               |  <NUMBER> <NUMBER> NORMAL

66   <TCONTROL END>  ::=  <START> <VERIFY> <OPERATE> <POINT LIST> <END>

67   <START>  ::=  <IDENTIFIER> IS-TYPE <IDENTIFIER>

68   <END>  ::=  . END

69   <OPERATE>  ::=  . OPERATE <NUMBER> AT <IDENTIFIER>
70                 |  . RELEASE <NUMBER> AT <IDENTIFIER>
71                 |  . OPERATE NULL
72                 |  . RELEASE NULL

73   <POINT LIST>  ::=  <POINT INFO>
74                    |  <POINT LIST> <POINT INFO>

75   <POINT INFO>  ::=  <IDENTIFIER> IS <IDENTIFIER> <NUMBER> <NUMBER>

76   <TCONTROL_SEQUENCE END>  ::=  <SEQ_START> <VERIFY> <ID LIST> . END

77   <SEQ_START>  ::=  <IDENTIFIER>

78   <VERIFY>  ::=  . VERIFICATION <PHRASE LIST>

79   <ID LIST>  ::=  <IDENTIFIER>
80                 |  <ID LIST> <IDENTIFIER>
```

APPENDIX II XPL Grammar

GRAMMAR ANALYSIS ---JM VERSION--- ANALYZER VERSION OF DEC 72

TODAY IS 01-26-73

P R O D U C T I O N S

```
 1   <PROGRAM>  ::=  <STATEMENT LIST> EOF

 2   <STATEMENT LIST>  ::=  <STATEMENT>
 3                       |  <STATEMENT LIST> <STATEMENT>

 4   <STATEMENT>  ::=  <BASIC STATEMENT>
 5                  |  <IF STATEMENT>

 6   <BASIC STATEMENT>  ::=  <ASSIGNMENT> ;
 7                          |  <GROUP> ;
 8                          |  <PROCEDURE DEFINITION> ;
 9                          |  <RETURN STATEMENT> ;
10                          |  <CALL STATEMENT> ;
11                          |  <GO TO STATEMENT> ;
12                          |  <DECLARATION STATEMENT> ;
13                          |  ;
14                          |  <LABEL DEFINITION> <BASIC STATEMENT>

15   <IF STATEMENT>  ::=  <IF CLAUSE> <STATEMENT>
16                     |  <IF CLAUSE> <TRUE PART> <STATEMENT>
17                     |  <LABEL DEFINITION> <IF STATEMENT>

18   <IF CLAUSE>  ::=  IF <EXPRESSION> THEN

19   <TRUE PART>  ::=  <BASIC STATEMENT> ELSE

20   <GROUP>  ::=  <GROUP HEAD> <ENDING>

21   <GROUP HEAD>  ::=  DO ;
22                    |  DO <STEP DEFINITION> ;
23                    |  DO <WHILE CLAUSE> ;
24                    |  DO <CASE SELECTOR> ;
25                    |  <GROUP HEAD> <STATEMENT>

26   <STEP DEFINITION>  ::=  <VARIABLE> <REPLACE> <EXPRESSION> <ITERATION CONTROL>

27   <ITERATION CONTROL>  ::=  TO <EXPRESSION>
28                          |  TO <EXPRESSION> BY <EXPRESSION>

29   <WHILE CLAUSE>  ::=  WHILE <EXPRESSION>

30   <CASE SELECTOR>  ::=  CASE <EXPRESSION>

31   <PROCEDURE DEFINITION>  ::=  <PROCEDURE HEAD> <STATEMENT LIST> <ENDING>

32   <PROCEDURE HEAD>  ::=  <PROCEDURE NAME> ;
33                         |  <PROCEDURE NAME> <TYPE> ;
34                         |  <PROCEDURE NAME> <PARAMETER LIST> ;
35                         |  <PROCEDURE NAME> <PARAMETER LIST> <TYPE> ;
```

```
36   <PROCEDURE NAME>  ::=  <LABEL DEFINITION> PROCEDURE

37   <PARAMETER LIST>  ::=  <PARAMETER HEAD> <IDENTIFIER> )

38   <PARAMETER HEAD>  ::=  (
39                       |  <PARAMETER HEAD> <IDENTIFIER> ,

40   <ENDING>  ::=  END
41               |  END <IDENTIFIER>
42               |  <LABEL DEFINITION> <ENDING>

43   <LABEL DEFINITION>  ::=  <IDENTIFIER> :

44   <RETURN STATEMENT>  ::=  RETURN
45                        |  RETURN <EXPRESSION>

46   <CALL STATEMENT>  ::=  CALL <VARIABLE>

47   <GO TO STATEMENT>  ::=  <GO TO> <IDENTIFIER>

48   <GO TO>  ::=  GO TO
49              |  GOTO

50   <DECLARATION STATEMENT>  ::=  DECLARE <DECLARATION ELEMENT>
51                             |  <DECLARATION STATEMENT> , <DECLARATION ELEMENT>

52   <DECLARATION ELEMENT>  ::=  <TYPE DECLARATION>
53                          |  <IDENTIFIER> LITERALLY <STRING>

54   <TYPE DECLARATION>  ::=  <IDENTIFIER SPECIFICATION> <TYPE>
55                        |  <BOUND HEAD> <NUMBER> ) <TYPE>
56                        |  <TYPE DECLARATION> <INITIAL LIST>

57   <TYPE>  ::=  FIXED
58             |  CHARACTER
59             |  LABEL
60             |  NESTED
61             |  EXTERNAL
62             |  <BIT HEAD> <NUMBER> )
63             |  <CHARACTER HEAD> <NUMBER> )

64   <BIT HEAD>  ::=  BIT (

65   <CHARACTER HEAD>  ::=  CHARACTER (

66   <BOUND HEAD>  ::=  <IDENTIFIER SPECIFICATION> (

67   <IDENTIFIER SPECIFICATION>  ::=  <IDENTIFIER>
68                                |  <IDENTIFIER LIST> <IDENTIFIER> )

69   <IDENTIFIER LIST>  ::=  (
70                        |  <IDENTIFIER LIST> <IDENTIFIER> ,

71   <INITIAL LIST>  ::=  <INITIAL HEAD> <CONSTANT> )

72   <INITIAL HEAD>  ::=  INITIAL (
73                     |  <INITIAL HEAD> <CONSTANT> ,

74   <ASSIGNMENT>  ::=  <VARIABLE> <REPLACE> <EXPRESSION>
```

```
 75                    |  <LEFT PART> <ASSIGNMENT>

 76   <REPLACE>  ::=  =

 77   <LEFT PART>  ::=  <VARIABLE> ,

 78   <EXPRESSION>  ::=  <LOGICAL FACTOR>
 79                    |  <EXPRESSION> | <LOGICAL FACTOR>

 80   <LOGICAL FACTOR>  ::=  <LOGICAL SECONDARY>
 81                          |  <LOGICAL FACTOR> & <LOGICAL SECONDARY>

 82   <LOGICAL SECONDARY>  ::=  <LOGICAL PRIMARY>
 83                             |  ¬ <LOGICAL PRIMARY>

 84   <LOGICAL PRIMARY>  ::=  <STRING EXPRESSION>
 85                           |  <STRING EXPRESSION> <RELATION> <STRING EXPRESSION>

 86   <RELATION>  ::=  =
 87                 |  <
 88                 |  >
 89                 |  ¬ =
 90                 |  ¬ <
 91                 |  ¬ >
 92                 |  < =
 93                 |  > =

 94   <STRING EXPRESSION>  ::=  <ARITHMETIC EXPRESSION>
 95                             |  <STRING EXPRESSION> || <ARITHMETIC EXPRESSION>

 96   <ARITHMETIC EXPRESSION>  ::=  <TERM>
 97                                 |  <ARITHMETIC EXPRESSION> + <TERM>
 98                                 |  <ARITHMETIC EXPRESSION> - <TERM>
 99                                 |  + <TERM>
100                                 |  - <TERM>

101   <TERM>  ::=  <PRIMARY>
102              |  <TERM> * <PRIMARY>
103              |  <TERM> / <PRIMARY>
104              |  <TERM> MOD <PRIMARY>

105   <PRIMARY>  ::=  <CONSTANT>
106                 |  <VARIABLE>
107                 |  ( <EXPRESSION> )

108   <CONSTANT>  ::=  <STRING>
109                 |  <NUMBER>

110   <VARIABLE>  ::=  <IDENTIFIER>
111                 |  <SUBSCRIPT HEAD> <EXPRESSION> )

112   <SUBSCRIPT HEAD>  ::=  <IDENTIFIER> (
113                          |  <SUBSCRIPT HEAD> <EXPRESSION> ,
$0
```

```
CPU TIME USED WAS 0.91 SECONDS.
TOTAL TIME IS 0.91 SECONDS.
```

REFERENCES

1. MacDonald Dettwiler and Associates, "B.C. Telephone System Builder", User's Manual, October 4, 1969.

2. Herbert E. Pike, Jr., "Process Control Software", Proc. IEEE, Vol. 58, pp. 87-97, January 1970.

3. W.M. McKeeman, J.J. Horning, D.B. Wortman, A Compiler Generator, Prentice-Hall Inc., 1970.

4. Peter Naur, "Revised Report on the Algorithmic Language Algol 60", Communications of the ACM, Vol. 6, pp. 1-17, January 1963.

5. D.E. Knuth, "On the Translation of Languages from Left to Right", Information and Control, Vol. 8, pp. 607-639, 1965.

6. Franklin L. DeRemer, "Simple LR(k) Grammars", Communications of the ACM, Vol. 14, pp. 453-460, July 1971.

7. C.J. Shaw, "JOVIAL - A Programming Language for Real-time Command Systems", Annual Review in Automatic Programming, 1963.

8. C. Gordon Bell, Allen Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971, p. 9.

9. J. Madderom, "User's Manual - A Translator Writing System for Mini-computers", UBC, 1973.

10. Jean-Pierre Levy, "Automatic Correction of Syntax Errors in Programming Languages", Ph.D. Thesis, Cornell University, December 1971.

11. A.J. Pieké, "Design and Implementation of an Interactive Graphics Language for Computer Aided Design", Masters Thesis, U.B.C., June 1973.