THE UNIVERSITY OF MANITOBA

A PASCAL COMPILER FOR IBM 360/370 COMPUTERS

by

W. BRUCE FOULKES

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

WINNIPEG, MANITOBA

October, 1975

"A PASCAL COMPILER FOR IBM 360/370 COMPUTERS"

by

W. BRUCE FOULKES

A dissertation submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

To my Parents

ACKNOWLEDGEMENTS

I gratefully acknowledge the many comments and suggested improvements to the thesis made by members of the examining committee. Finally, I am indebted to my supervisor, Professor James M. Wells, for his encouragement and many valuable suggestions on all facets of the thesis.

ABSTRACT

This thesis describes a compiler to translate the programming language PASCAL into OS-compatible object modules for the IBM 360/370. The compiler employs an unusual design strategy and is rare among PASCAL compilers in that it is not a direct descendent of the original compiler developed for CDC machines. The compiler is intended for use in a production environment and therefore stresses the areas of fast compilation, compile-time error checking, run-time error detection and the production of efficient object code. Compilation strategy, internal organization and code generation are discussed in detail. Examples demonstrating the features and performance of the compiler are included.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This thesis is divided into seven chapters. The current chapter presents a brief history of the development of the language PASCAL, and the original implementation for it. The objectives of the thesis are then indicated. Chapter 2 describes the related work in the area. Particular emphasis is given to the other implementations which have been attempted, and the burgeoning literature on the subject. Chapter 3 describes the implementation methods which were considered and the one which was adopted, and describes the results achieved in the light of similar attempts. Examples are given demonstrating compile-time and run-time error checks, and a large example is included which demonstrates many of the features of the language. Chapter 4 serves as a User's Manual and describes language constructs with reference to this implementation. Chapter 5 details the internal organization of the compiler, with particular emphasis on the design and use of the symbol table and other internal tables. The run-time organization and register usage are also described. Chapter 6 demonstrates the code generated for most constructs of the language. Run-time checking code and code optimization techniques are included in the examples. Chapter 7 presents conclusions on the project and outlines areas where further work is considered to be necessary.

The format of the thesis is intended to provide suitable break points for readers of varying degrees of interest. Reading to the end of Chapter 3 gives the historical background, an insight into the literature, a description of the approach used and the results achieved demonstrated with sample programs. If the reader is interested in using the compiler, continuing to the end of Chapter 4 provides the necessary information about language constructs. A reader who desires more detailed information about the internal organization of the compiler and the code which is generated should read to the end of Chapter 6.

1.1 HISTORICAL BACKGROUND

In recent years, George H. Richmond, of the University of Colorado has undertaken the task of editing the "Pascal Newsletter"; hereafter referred to as the Newsletter. To date, three Newsletters have been issued; No. 1 in January 1974 (Ric74a), No. 2 in May 1974 (Ric74b) and No. 3 in February 1975 (Ric75). The express purpose of the Newsletter is "to keep the PASCAL community informed about the efforts of individuals to implement PASCAL on different computers and to report extensions made to the language". In Newsletter No. 3, Richmond presents a history of PASCAL. The history of the development of PASCAL and the implementations on the original CDC machines is therefore presented with suitable excerpts from this source.

> "Pascal is an outgrowth of the early efforts of Dr. Niklaus Wirth to define a successor to Algol 60 by extension into the area of data definition facilities. In 1965, a proposal for such a successor was presented to an IFIP Working Group. It was

deemed not to be a sufficient advance over Algol 60 and was released for wider publication and appeared in the Communications of the Association for Computing Machinery (Wir66). Some of the aims of this proposal were to define a language that could be compiled quickly by a reliable translator, extend the utility of the language beyond strictly algebraic and numeric applications into areas such as information retrieval and symbol manipulation, and to oblige the programmer to express himself clearly without subverting the language to accomplish his task.

"These goals reappear in the original definition of the language Pascal (Wir71a). There was a desire to provide a language well suited to teaching programming as a systematic discipline based on fundamental concepts that were visibly reflected in the language. And Dr. Wirth wanted to demonstrate through an actual implementation that a significant and useful language can be realized in a reliable fast compiler.

"The first work on Pascal began in 1968 at Eidgenossische Technische Hochschule (ETH) in Zurich, Switzerland on a CDC 6000 system with the construction of a Pascal compiler written in Fortran (Wir71b). Although the compiler was completed, the result was an internal data structure and program contorted to fit the rules of Fortran. This compiler was unsuitable for translating at the source level into Pascal and a new compiler was started. It was entirely written in Pascal.

"Once the project had progressed to the point that the paper compiler could theoretically translate itself, the compiler was translated by hand into a low level language and an executable compiler was produced. This hand translation only took one man-month of effort. Admittedly, only enough of the language facilities to allow self-compilation had been implemented, but 60 percent of the final version was already there. Almost 3000 lines of code were produced before testing could begin. The full compiler was available in 1970 and this version is called PASCAL0.

"....The original report on Pascal provided a rigorous definition of the syntax of the language but only a verbal description of the semantics. The semantics were provided in 1972 by the publication of an axiomatic definition of Pascal (Hoa73). As part of this project, the syntax of the language was revised slightly and the revised report on Pascal was issued (Wir72a). To bring the existing compiler up to date, the necessary cosmetic changes, except for class variables which were retained from the previous version, were made and a new compiler was released. This version is refered to as PASCAL1 and was first available in late 1972. It is also known as PASCAL 6000-3.2.

> "However, the decision was made to rewrite the compiler completely. The changes included replacement of class variables with pointer variables to conform with the revised report, relocatable binary object code compatible with the standard CDC loader instead of a special absolute loader, introduction of a facility for separately compiled procedures and external Fortran subroutines, and new transput procedures for eliminating the need for a line control character and to accomodate CDC disk file structures. The new Pascal compiler was released in June 1974 and is called PASCAL2. It is also known as PASCAL 6000-3.4."

1.2 OBJECTIVES

The language PASCAL offers many advantages over the other languages in wide use. It has a simple, straightforward syntax which does not require complex and time-consuming parsing schemes. The statements offered by the language are easy to use and allow the writing of programs in a structured manner. The data types are especially appealing, allowing the definition of complex data structures, along with the introduction of scalar types and sets. The language is designed to be reasonably transportable between machines, and this is considered to be a great advantage.

The language was originally implemented on a CDC machine. For the language to gain wide acceptance, especially in North America, a compiler for IBM machines was a necessity.

It was therefore decided to write a PASCAL compiler for the IBM 360, which would not merely be an academic exercise, but would be usable in a production manner. It was decided to implement the language as defined in the Report (Wir71a, and later Wir72a), in the interests of portability. The temptation to introduce improvements

and modifications was therefore resisted in all but a few cases.

Once the decision was made to implement PASCAL, several considerations had to be taken into account before deciding on the implementation route.

It seemed that the principal users of PASCAL in the foreseeable future would be academic institutions. For a language and compiler to be used widely in a student environment, expecially at the undergraduate level, special conditions had to be met.

The compiler had to perform well when presented with source programs containing errors, and had to produce meaningful error messages. Errors in program logic, which do not become apparent until run time, had to be detected as soon as possible so that an indication of the problem could be given.

Another consideration in whether or not to use a language (and compiler) is the cost. This is principally a function of the memory size required by the compiler and compile speed. If the memory requirement is unusually high, this consideration alone can limit its use at some installations.

It was therefore decided to make the major design objectives the following:

1 - to perform exhaustive compile-time checks, taking full advantage of the redundancy of the language.

2 - to recover from syntax errors in the source as soon as possible so that meaningful error checking of the remainder of the source can be performed.

3 - to generate run-time checking code so that meaningful run-time error messages can be given, but at the same time allow this checking code to be suppressed if desired for increased efficiency in production runs.

4 - to make the compiler as fast as possible, and still meet the previous design objectives.

5 - to keep the memory requirements as low as possible, and still meet the other design objectives.

CHAPTER 2

RELATED WORK

2.1 IMPLEMENTATIONS

Since the original implementation of PASCAL on the CDC 6000 series, the language has been implemented on almost every major machine. The Newsletter reports that implementations have been accomplished on at least the following: CII IRIS 80, CII 10070, XDS SIGMA 7, DEC SYSTEM 10, IBM 360/370, UNIVAC 1108, ICL 1900 series and MULTIM. In addition projects are underway for: TI ASC, Data General 840, Raytheon 704, Univac AN/VYK-20, Honeywell G635, Burroughs 4700 and 6700 and the PDP-11/45.

The first transfer of PASCAL to another machine was accomplished by Drs. J. Welsh and C. Quinn of Quenn's University in Delfast, North Ireland, who transported PASCAL to the ICL 1900 Series Computers (Wel72). Their method was to change the code generation segment of the original compiler so that it produced ICL 1900 object code. A simulator for the ICL 1900 was written in PASCAL and run on the CDC 6000 at Zurich to test the resulting code. Careful preparation of the source programs plus an intensive short-term effort in the summer of 1971 resulted in the successful transfer of the compiler.

A compiler for IBM 360/370 computers has been produced at Grenoble, France. No documentation on this project seems to be available but it is believed that the technique used is the

following (Tas). The CDC compiler, written in PASCAL, was translated into PL/1, and the code generation portions were modified so that 360 Assembler is produced. A second compiler, written in PASCAL, was then compiled using the compiler written in PL/1, thus bootstrapping the language onto the IBM 360.

Mr. D. Russell and Mr. J. Sue have completed the bootstrap of a PASCAL compiler written in PASCAL on the IBM 360/370. The language implemented is a subset with just enough features to allow the compiler to compile itself. This implementation is based on the CDC 6600 compiler of January 1972 (Ric75).

Another method, besides the bootstrap technique, of transporting PASCAL to other machines, became available, and since has come into wide usage. This is the interpretive approach. Richmond in Newsletter No. 3 describes this development as follows:

> "During the past two years, another method for implementing PASCAL on computers other than the CDC 6000 series has been available from ETH. A portable Pascal compiler was released in early 1973 for this purpose. This first version is called PASCAL-P1. It was written to generate in an abstract language called P-code. By compiling the compiler, a P-code Pascal compiler is obtained. Thus, one only need provide an interpreter for P-code in order to obtain a Pascal compiler. However, the PASCAL-P1 compiler followed the original definition of Pascal (PASCAL0 compiler) and did not implement all the features available in the CDC 6000 version.
>
> "To remedy this problem, the portable Pascal compiler was rewritten and released in November 1974. It is called PASCAL-P2. The new compiler can be tailored to the target machine by specifying several parameters...."

This approach is being used by many people to accomplish a rapid implementation of PASCAL on other machines.

Mr. Alfred C. Hartmann and Robert S. Deverill, of the California Institute of Technology have implemented a subset of PASCAL using PASCAL-P1 (Dev74). At the present time, this system seems to be the one in widest use on IBM machines.

Other interpretive PASCAL projects for IBM machines have been reported by Dr. John Larmouth at Cambridge, England, and Dr. S. V. Rangaswamy at Bangalore, India (Ric75).

2.2 LITERATURE

Since PASCAL was originally defined (Wir71a), a considerable amount of literature has been written on the subject. Further language specifications followed: the Revised Report (Wir72), an axiomatic description of the language (Hoa73), and finally a user's manual (Jen74).

The Pascal Newsletter (Ric74a, Ric74b and Ric75) is playing an invaluable role in distributing information about further implementations of the language.

The language has been criticized by some people and praised by others. The most direct assault on the language was by A. N. Habermann (Hab73) who attempted to demonstrate that there are features in PASCAL which can cause problems if used in certain ways. This criticism was countered by Olivier Lecarme and Pierre Desjardins (Lec74b) who demonstrate that most of the problems indicated by Habermann were through poor programming practice, and state that it is possible to write poor programs in any language.

Many people consider PASCAL an ideal introductory language. Wirth uses PASCAL for his examples in his book on systematic programming (Wir73). Lecarme makes the case that PASCAL is an ideal language for teaching structured programming (Lec74a). This work includes an excellent bibliography with over a hundred references which serves as a very complete literature survey on these topics.

CHAPTER 3

IMPLEMENTATION STRATEGY

3.1 STRATEGIES CONSIDERED

As indicated in Chapter 2, several implementation strategies are possible. When work on this project was begun, the PASCAL-P approach was not available. It is doubtful that this approach would have been adopted in any event, as the resultant high core requirement and slow compile speed are inconsistent with the design objectives. Under the heading "PASCAL and Portability" in Newsletter No. 2 (Ric74b), Niklaus Wirth, the designer of the PASCAL language, states on this subject:

> "The Pascal-P approach is quite adequate and convenient, if efficiency of program execution is of no great significance. However, if the development of a high-quality compiler is the objective, a bootstrapping process on the basis of an interpretive Pascal-P system is very costly, and involves a large amount of reprogramming. It is clear that a different approach to the transportation of compilers themselves should be investigated."

The method of Welsh and Quinn (Wel72) was very attractive from the point of view of the time necessary to accomplish the transfer. The prime reason for not adopting this approach was the lack of availability of a CDC machine.

The method used at Grenoble (Tas) of translating the CDC PASCAL compiler into PL/1 was considered, but it was felt that the resulting compiler would not be as fast as it could be if it was written in a lower-level language.

The architecture of the CDC and IBM machines differs considerably and it was felt that any attempt to transport the compiler from the CDC to IBM machines, by either of the strategies discussed above, would lead to problems and in particular inefficiencies. If this approach was taken, the compile-time checking and error recovery would be dictated by that performed on the CDC compiler. In addition, modifying the code generators for another machine would not allow object code optimization to be accomplished as readily as if the compiler was designed from scratch for the target machine.

For these reasons it was decided to adopt an entirely different approach and write the compiler completely independently of the original PASCAL compiler for CDC machines.

3.2 STRATEGY ADOPTED

At the time the decision to write a PASCAL compiler was made, Rainer Kossmann was developing a Translator-Writing System called SYNTICS (Kos72), which he intended to use as a tool in the development of an ALGOL68 compiler. SYNTICS was the successor of an earlier system called RSYN (Kos72), which was PL/1 based. The RSYN system had been used by the author in the joint development, with Kossmann, of an interpreter for a logic language called DECLAB (Fou72).

SYNTICS is a PL360-based (Mal71 and Wir68) Translator-Writing System using a left to right, top-down parsing strategy. It accepts a modified BNF description of the syntax of a programming language, and optionally produces tables for a PL360 table-driven parser, or

generates an Assembler language parser. A scan routine is provided to read the source and present it to the parser in an acceptable form. Built-in phrases (BIPs) are pre-defined and allow the user to specify often used semantic actions, such as "look for an identifier", by simply using the name of the BIP. SYNTICS allows the user to specify points in the parse at which semantic actions are to take place. The names of procedures which are to perform the semantic actions (referred to as control phrases) are included in the desired places in the syntax description. These routines are then called when the indicated locations in the parse are reached. A facility for producing an analysis record of the parse is provided for multi-pass compilers. This analysis record consists of a tree-structured trace of the parse and can serve as the intermediate form between compiler passes. At present this feature is available only with the table-driven parser.

It was decided to use the SYNTICS system for the following reasons. The feature of being able to specify semantic actions at any desired place during the parse is very useful for early error detection during compilation. This allows an error message to be specified at precisely the point in the parse at which the error is detected.

The separation of syntax and semantics allows recovery for programs containing syntax errors to be handled almost entirely with syntax specifications. This means that changes can be made to the syntactic error recovery scheme without having to worry about possible side effects in the semantic routines.

Many trace features were provided, such as allowing a complete trace of the parse, which proved very useful especially during the design of the syntactic error recovery.

A compiler can be written in an extremely modular manner with different procedures called to handle the semantic actions at different places during the parse. This enables the semantic actions to take place at any time, to be well defined, and allows many semantic routines to be tested independently of other parts of the compiler.

Many of the BIPs and scan routine, etc. are common to any compiler and it seemed that much duplication of effort could be eliminated by using those routines which had already been developed.

Another major consideration was that the SYNTICS system was at a very early state of development and the possibility existed to influence the design to provide a more rounded system, which would have a wider applicability than if it was just designed to handle ALGOL68.

As Algol W (Wir66 and Bau68a) has many similarities to PASCAL, and was also implemented (Bau68b) using PL360, this compiler was investigated for common areas. Some ideas and code were actually adopted in the areas of code generation, arithmetic functions and input/output routines.

D. Moir wrote routines to allow formatted input/output in Algol W (Moi71), and some code was borrowed from this source.

3.2.1 FORM OF THE SYNTAX

The SYNTICS system requires that the syntactic description of the grammar be provided using the following notation. Terminals, nonterminals, BIPs and control phrases are all enclosed in angular brackets. Terminals are preceded by the symbol @ while BIPs are indicated with $ and control phrases are preceded by an *.

The main BIPs used in the PASCAL system are the following:

$IDENT - identifier
- a check is made to ensure that it is not a reserved word in the language.

$INT - integer

$DENOTATION - any of integer, real, string, TRUE, FALSE, hexadecimal constant etc.

$INTLABEL - integer treated as a character string.

When one of the above BIPs is specified, a system-provided semantic routine is called which looks in the source for the required construct. If it is found, the entry is hashed and SUCCESS is reported to the parser. If the construct is not found, the BIP reports FAILURE.

Control phrases also return either SUCCESS or FAILURE to the parser. Most control phrases just perform some operation, such as making an entry on the symbol table or generating code for some construct and therefore return SUCCESS when complete. A few routines are used to make decisions, such as whether or not an identifier is defined; these return SUCCESS or FAILURE depending on the test and thus have an effect on the direction taken by the parser.

Example:

```
- - <$IDENT> ( <*DEFINED> <OPTION1> | <OPTION2> )
```

The BIP $IDENT is called which searches in the source starting at the current pointer for an identifier. An identifier, as implemented in PASCAL, starts with a letter, followed by a sequence of letters, digits or underscores. If what appears to be an identifier is found, it is hashed to determine if it is in fact an identifier or a reserved word in the language. $IDENT returns SUCCESS only if the required token is found and is not a reserved word. The brackets after $IDENT indicate alternatives. The first alternative begins with a call on the control phrase DEFINED. This PL360 procedure uses the hash address provided by $IDENT to check if the identifier is currently defined. If it is, DEFINED reports SUCCESS and the parser proceeds to look for OPTION1. If DEFINED reports FAILURE the parser skips to the next alternative and looks for OPTION2.

Several other BIPs are used to control the syntactic error recovery. The recovery mechanism consists of a search for reserved words or other terminal symbols which mark distinctly recognizable points at which the parser can be reasonably sure of recovering. This parse recovery mechanism is highly recursive and the parser can get nested to many levels in the recovery part of the syntax. This mechanism was originally used on the SYNTICS system for lookahead when parsing ALGOL68 programs but proved useful for syntactic error recovery as well.

The following BIPs are used in the syntactic error recovery:

$SETLT - sets lookahead flag and reports SUCCESS.

$SETLF - sets lookahead flag and reports FAILURE.

$IFL - interrogates lookahead flag and reports SUCCESS if it is currently set; otherwise it reports FAILURE.

$IFNOTL - interrogates lookahead flag and reports SUCCESS if it is not currently set; otherwise it reports FAILURE.

Both $IFL and $IFNOTL reset the lookahead flag after interrogating it.

Example of Recovery Syntax:

```
                - - - ( <CONSTRUCT> | <ERROR<0,*>> );
<ERROR> := <$IFNOTL> <LOOKFOR>;
<LOOKFOR> := ( <@TERM1> | <@TERM2> | <@TERM3> ) <$SETLF>
           | <$IFL> <$SETLT>
           | <SCANTONEXTSYMBOL>;
```

The <0,*> means any number of occurrences including none. <ERROR> is looked for repeatedly until it returns FAILURE. <$SETLF> is used to set the lookahead flag and by reporting FAILURE causes the parser to backtrack the pointer before the terminal which was found. <$IFL> tests the lookahead flag. If set, <$SETLT> sets the flag again and reports SUCCESS for <ERROR>. <$IFNOTL> returns FAILURE on the next attempt ending the recovery sequence. If none of the terminals specified is found, <SCANTONEXTSYMBOL> skips the input pointer to the next terminal in the source and the process is repeated.

3.3 DETAILS OF PASCAL COMPILER

The PASCAL compiler is one-pass. Another pass would have made compile-time checking and code generation a little easier and might have decreased the core requirements, but would have had an adverse effect on compile speed, with the necessary production of an intermediate representation.

The table-driven parser was used during the early phases of compiler design. This provides many trace facilities and is inexpensive to compile. The generated Assembler parser takes a minute of CPU time to assemble and this is too costly to use when the syntax is changing often. These changes to the syntax were not in the form of modifications to the language, but merely changes to the description given to the SYNTICS system. Control phrases were inserted as the semantic routines were written and additional specifications were included to control the syntactic error recovery.

The Assembler parser is much faster than the equivalent table-driven parser and for this reason it was decided very early to use this parser in the final production version. The compiler with the Assembler parser is approximately 15K larger than, but runs at 3 times the speed of, a version with the table-driven parser. The fact that an analysis record is not available with the Assembler version of the parser was also a consideration in the decision to perform the analysis in one pass.

The syntactic error recovery mechanism used provides many levels of recovery from coarse to extremely fine. Its inclusion doubled the