IMPLEMENTATION OF A COMPILER FOR THE FUNCTIONAL PROGRAMMING LANGUAGE PHI(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA E J COLE ET AL. JUN 87
THESIS

IMPLEMENTATION OF A COMPILER FOR THE
FUNCTIONAL PROGRAMMING LANGUAGE SHL - I

by
Eugene J. Cole
and
Joseph E. Donnell II

June 1987

Thesis Advisor: Daniel Davis

Approved for public release; distribution is unlimited.
This report describes the design and implementation of a prototype compiler for the functional programming language F5. The design is derived from an algorithmic approach, and the authors think this should facilitate the understanding of the design and implementation. The front-end of the compiler includes an independent lexical and syntactic analyzer, type-checking routines, and tree-structured data representations. The back-end implements a machine-independent semantic analyzer and code generator.

Since this implementation is a prototype, it does not support all the features available in a full implementation. The basic constructs of F5, such as functions and data definitions, are implemented, as well as conditionals, arithmetic, relational, and boolean types. However, the necessary features are present and the design is mature enough to allow expansion to a full implementation.

**Abstract Security Classification:** Unclassified

<table>
<thead>
<tr>
<th><strong>Subject Keywords</strong></th>
<th>Functional Language; Applicative Language; Compiler Design</th>
</tr>
</thead>
</table>

**Program Element No.** | **Project No.** | **Task No.** | **Work Unit No.** |
|-------------------------|-----------------|--------------|------------------|

**Report Date:** 1977 June

**Address:** Monterey, California 93943-5000

**Office Symbol:** Code 92

**Monitoring Organization:** Naval Postgraduate School

**Name of Performing Organization:** Naval Postgraduate School

**Address:** Monterey, California 93943-5000

**Name of Funding Sponsor:** Naval Postgraduate School

**Address:** Monterey, California 93943-5000

**Sponsor's Name:** Cole, Burke J. and Connell, Joseph E. III

**Sponsor's Code:** 000

**Related Work:**

- Functional Language; Applicative Language; Compiler Design

**Year:** 1977

**Month:** June

**Day:** 30

**Place:** Monterey, California

**Abstract Security Classification:** Unclassified

**Approved for Public Release:** Distribution is Unlimited

**Distribution Availability of Report:**

**Monitoring Organization Report Number(s):**

**Performing Organization Report Number(s):**

**Office Symbol:**

- **Name of Monitoring Organization:** Naval Postgraduate School
- **Address:** Monterey, California 93943-5000
- **Office Symbol:** Code 92

**Name of Funding Sponsor:** Naval Postgraduate School

**Address:** Monterey, California 93943-5000

**Name of Performing Organization:** Naval Postgraduate School

**Address:** Monterey, California 93943-5000

**Office Symbol:** Code 92

**Date of Report:** 1977 June

**Fiscal Year:** 1977

**Fiscal Quarter:** 3rd

**Fiscal Month:** June

**Fiscal Day:** 30

**Place:** Monterey, California
Implementation of a Compiler for the Functional Programming Language PHI — Φ

by

Eugene J. Cole
Major, United States Marine Corps
B. A., The Citadel, 1975

and

Joseph E. Connell II
Captain, United States Marine Corps
B. S., University of Missouri — Rolla, 1974

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

Authors:  

Eugene J. Cole

Joseph E. Connell II

Approved by:  

Daniel Davis, Thesis Advisor

Bruce J. MacLennan, Second Reader

Vincent Y. Lam, Chairman,
Department of Computer Science

Kneale T. Marshall,
Dean of Information and Policy Sciences
ABSTRACT

This thesis describes the design and implement of a **prototype** compiler for the functional programming language PHI. The design is highly modularized and the authors think this should facilitate the understanding of both concept and implementation. The front-end of the compiler implements machine independent lexical and syntactic analyzers; top-down parsing techniques are employed. The back-end implements a machine dependent one-pass semantic analyzer and code generator.

Since this implementation is a **prototype**, it does not possess all of the qualities desirable in a full implementation. The basic constructs of PHI: functions and data definitions are implemented, as well as the integer, natural number, and boolean types. However, the necessary hooks are present and the design is mature enough to allow expanding the prototype to a full implementation.
# TABLE OF CONTENTS

I. **INTRODUCTION** ........................................................................................................... 6  
   A. **BACKGROUND — GENERAL** ................................................................................. 6  
   B. **BACKGROUND — THESIS** .................................................................................... 7  
   C. **BACKGROUND — FUNCTIONAL PROGRAMMING** ............................................... 8  
      1. Problems with Conventional Languages ................................................................. 9  
      2. Functional Languages ............................................................................................. 10  
   D. **ASSUMPTIONS** ....................................................................................................... 11  
   E. **CONSTRAINTS** ......................................................................................................... 12  

II. **FRONT-END OF THE COMPILER** ............................................................................. 13  
    A. **LEXICAL ANALYSIS — THE SCANNER** ................................................................. 13  
    B. **SYNTACTIC ANALYSIS — THE PARSER** .............................................................. 18  
    C. **ERROR HANDLING** ............................................................................................... 23  

III. **BACK-END OF THE COMPILER** ............................................................................. 25  
    A. **OVERVIEW** ........................................................................................................... 25  
    B. **RUN-TIME ORGANIZATION** ............................................................................... 25  
    C. **SEMANTIC CHECKING AND CODE GENERATION** ............................................. 31  
    D. **OPTIMIZATION** ................................................................................................... 40  

IV. **RESULTS & CONCLUSIONS** ................................................................................. 42  
    A. **RESULTS** ............................................................................................................... 42  
    B. **CONCLUSIONS** ..................................................................................................... 43  

V. **FURTHER RESEARCH** ............................................................................................ 44  

**LIST OF REFERENCES** ................................................................................................. 46
I. INTRODUCTION

A. BACKGROUND — GENERAL

In its attempt to provide students with a well rounded background to the field of computer science, the computer science department at the Naval Postgraduate School offers courses covering recent developments in programming languages. One of the courses deals specifically with the methodology of functional, also known as applicative, programming. Both the theory and the practice of functional programming are covered, concentrating more on the practice than the theory. In order to fully appreciate the nuances of functional programming it would be desirable to provide the students with a functional programming environment. This would provide a first hand look at the fundamental difference in methodologies when programming in functional languages as opposed to programming in traditional imperative languages.

Of the languages currently supported in the department; LISP, on the UNIX\textsuperscript{1} environment, comes the closest to meeting this requirement. Although LISP is considered a functional language by some, its many extensions and modifications actually brings it into the world of imperative programming. It is not a pure functional programming language.

There are several additional problems associated with using LISP to teach techniques of functional programming. Modern LISP dialects do not support all aspects of functional programming. Most notably they lack the ability to define higher-order functions. Dynamic scoping and the semantics of the language make it a pedagogical nightmare to teach.[Ref. 1:p. \textsuperscript{0-1}] The goal of teaching functional programming would rapidly be overtaken by the necessity of explaining the idiosyncrasies of LISP. In an 11 week

\textsuperscript{1}UNIX is a trademark of Bell Laboratories.
quarter, time devoted to LISP would significantly detract from instruction of functional programming.

Recognizing the shortcomings of LISP, a pure functional language, PHI was developed by Dr. B. J. MacLennan for use in this course of instruction. The syntax of PHI closely follows that of standard mathematical notation. This means students should have little difficulty in learning how to write legitimate PHI statements. Instruction can now concentrate on joining these statements to create functional programs. Hopefully, this will lead to a greater understanding and appreciation of the methodology of functional programming.

B. BACKGROUND — THESIS

Creation of PHI solved the problem of finding a suitable language to use to demonstrate the methodology of functional programming. However, currently PHI programs are programs on paper only. There exists no programming environment for the PHI language. So it is impossible to machine execute PHI programs. This thesis attempts to remedy the above problem by providing the first component in a PHI programming environment — a prototype PHI compiler.

Conventional compiler construction techniques were chosen for this implementation for several reasons. By choosing conventional techniques, the authors were able to address the problems associated with utilizing conventional methods for implementing a compiler for a functional language. Additionally, realizing that both the language and system would change, the authors wanted a well documented and understood methodology. The cost of maintaining a system can be as much as three times the development cost [Ref. 2:p. 478]. Therefore, it was imperative to choose a methodology that supported a clean and structured design.

specific problems and solutions are covered later in Chapters Two and Three
Following conventional methodologies, the authors separated the PHI compiler design into a front-end and a back-end. The overall general design of the PHI compiler is shown in Figure 1.1. The front-end, containing the scanner (lexical analyzer) and parser (syntactic analyzer) is essentially responsible for analysis of the external file containing the source program. The PHI compiler back-end couples semantic analysis with code generation to produce code suitable for execution on the target machine. The authors felt that a clear and distinct separation between parts would aid understanding of the system, simplify division of labor, and increase ease of development and maintenance. It should also result in greater flexibility for follow-on development in the PHI programming environment. As an example, the current front-end could be modified to support a PHI interpreter.

![Figure 1.1 General Design of the PHI Compiler](image)

C. BACKGROUND — FUNCTIONAL PROGRAMMING

Functional programming is a methodology in favor among academicians. Although applicative programming goes further back, it is generally agreed that, as a methodology, functional programming traces its roots to John Backus. In

---

3Design and implementation of the front-end is discussed in Chapter Two.

4Design and implementation of the back-end is discussed in Chapter Three.
his acceptance speech for the 1977 ACM Turing Award, Backus criticized traditional programming languages and programming styles. He went on to propose a new methodology of programming that involved "the use of a fixed set of combining forms called functional forms." [Ref. 6:p. 619] This methodology is known today as functional programming.

1. Problems with Conventional Languages

Backus feels [Ref. 6:pp. 613–619] that the basic underlying problem with conventional languages is the existence of the assignment statement. The assignment statement plays a central role in conventional languages and breaks programming into two worlds. Backus calls the right-hand side of assignment statements, expressions, the first of these worlds. The second world is the world of statements, with the primary statement, of course, being the assignment statement.

Several problems are associated with assignment statements. First, they permit programs to be held hostage through access to their variables. Since variables are used to imitate the machine's storage cells, assignment statements allow, even encourage, state changes to take place. This access, either direct or indirect, permits such problems as side effects, unintentional state changes, and aliasing to arise. It then becomes difficult to reason about the correctness of these programs, so proving simple programs correct is an arduous task and proving complex programs correct is virtually impossible. Additionally, by permitting the value of variables to be changed, the assignment statement makes temporal order of execution of statements critical. For example, the following two pieces of code produce dramatically different results depending on which statement inside the for loop is executed first.

```c
for (i = Ø; i != some_value; ++i)
    { if (i % 2 == Ø)
        continue;
        DoSomething(i);
    }

for (i = Ø; i != some_value; ++i)
    { DoSomething(i);
      if (i % 2 == Ø)
        continue;
```

These problems interact so that it becomes extremely difficult to create new programs out of old ones. [Ref. 6:pp. 613 - 619, Ref. 1:pp. 1-2 - 1-20]

Another problem associated with assignment statements is that each produces only a one-word result. In effect, they force programmers to think in a word-at-a-time manner. For example, to apply a function to an entire array of values, the programmer must access each value individually. Not only is this wasteful of computer assets, but it results in what Backus refers to as the "von Neumann bottleneck" of conventional programming languages. [Ref. 6:pp. 613 - 619]

2. Functional Languages

Backus proposes the methodology of functional programming as the solution to these problems. Functional languages have removed variables and the assignment statement from their syntax so that their basic building block becomes the function. It is through "the use of a fixed set of combining forms...plus simple definitions" [Ref. 6:p. 619] that the programmer is able to build new functions from existing functions. It thus becomes possible to form a new program by combining two or more existing programs or functions together.

The absence of assignment statements and variables removes the problems plaguing conventional languages caused by side effects, etc. because the program now operates exclusively in the world of expressions. This permits the programmer to maintain a clear conceptual view of the program. It is easier to understand and reason about the task the program is to perform [Ref. 5:pp. 65 - 69]. It now becomes not only possible, but practical to prove programs correct [Ref.6:pp. 624 - 625].

Another direct benefit stemming from the absence of side effects is order. The values of expressions are no longer dependent on the order in which they are evaluated. Therefore, functional languages provide a natural means of performing parallel computations [Ref. 7:p. 35]. Functional languages and the associated methodology of
functional programming may very well provide the key to programming the massively parallel computers entering service nowadays. All of the above benefits have applicability to ongoing research in the SDI program.

The authors feel that functional programming can best be summarized by the following thought — assignment statements are to functional programming what GOTO statements are to structured programming.

D. ASSUMPTIONS

An IBM\textsuperscript{5} personal computer/IBM compatible personal computer was chosen as the target machine for this implementation. The authors felt that the nature of the language and its intended use were better suited for the PC/personal work station environment as opposed to a mini- or main-frame time shared environment. The PC environment should provide greater flexibility and freedom when implementing follow-on tools for the PHI programming language. Also, future compiler improvements will not have to be concerned with extraneous interfaces to another system. Working with a PC environment eliminates the need to take into account the effects the PHI environment will have on another user of the system. The implementor is able to work with a system that remains constant — a known quantity.

The assumed target machine configuration is based on the equipment available in the Naval Postgraduate School's computer science microcomputer lab. Each machine is configured with 640K bytes of RAM, one (most have two) 20M byte hard disk drive, one 1.2M byte 5 inch floppy disk drive, and the 8087 math co-processor; each currently operates under the MS-DOS\textsuperscript{6} 3.x operating system. These machines are readily available to all computer science students at the Naval Postgraduate School, and many students own

\textsuperscript{5}IBM is a registered trademark of Internal Business Machines Corporation.

\textsuperscript{6}MS-DOS is a registered trademark of Microsoft Corporation.
personal computers with similar configurations. It is not necessary to utilize a hard disk when executing the PHI compiler.

E. CONSTRAINTS

As is the case with most implementation theses, time was probably the biggest constraint facing the authors. This involved making certain trade-offs; e.g. should the major effort be directed towards a full implementation of PHI while concentrating on a particular component of the compiler, or should the major effort be directed towards a full implementation of the compiler while concentrating on a subset of the PHI language? The authors felt that the greatest benefit could be gained by implementing a complete compiler. Having to actually face the issues and problems associated with designing, implementing, and interfacing a full compiler implementation would be much different than just reading about them in a text. As a result, this thesis implements only a subset\(^7\) of PHI.

Since PHI is an experimental language it is still undergoing changes and revisions. Trying to modify and update the compiler design with each version proved to be an impossibility. The authors were forced to freeze the design based on the language as it stood on 07 January 1987. Any follow-on work will need to update the front-end and back-end of the compiler to meet the requirements of these new versions of PHI. A description of the grammar as implemented and a description of the latest version of the grammar may be found in the Appendixes.

\(^7\)This subset is discussed in the individual chapters on the front-end and back-end.
II. FRONT-END OF THE COMPILER

The authors separated the design of the PHI compiler into two modules, a front-end and a back-end. These modules were then further subdivided to produce the general layout of Figure 1.1. The authors believe this modularization simplifies the design and will aid in understanding the system, thus decreasing future maintenance problems.

The front-end of the PHI compiler is comprised of the scanner (lexical analyzer), the parser (syntactic analyzer), and their associated error recovery routines. Two possible interactions between the lexical and syntactic analyzers were considered. The first incorporates the scanner into the parser, and tokens are produced by the scanner only upon request of the syntactic analyzer. Thus, this system acts like a pipeline. An alternate method is to allow the scanner to tokenize the entire source program, store the tokens in some data structure, and pass this structure to the parser. [Ref. 3:p. 10]

For the prototype implementation of a PHI compiler, the authors based the design on the first interaction. Although the second method is conceptually very easy to understand, the authors think the current implementation is clean and will readily lend itself to future enhancements. Any input alphabet peculiarities are restricted to the lexical analyzer, and this independence should provide benefits for the next student(s) who work on the PHI programming environment.

A. LEXICAL ANALYSIS — THE SCANNER

The PHI compiler reads a source file of ASCII text which is fed to the scanner for lexical analysis. The principle task of lexical analysis is to separate or divide the source program into tokens for use during syntactic analysis [Ref.8:p. 84, Ref. 9:p. 155]. This is accomplished in the PHI compiler through a character-by-character examination of the
user's source file. These characters are assembled/grouped into the individual tokens which represent terminal symbols of the PHI grammar. Examples of some of the terminal symbols are operators, identifiers, keywords, and constants. A complete listing of the PHI tokens may be found in the header file for the scanner in Appendix E.

The primary advantage to tokenizing the source program is that the design of the syntactic analyzer needs to take into account only one type of data unit — the token [Ref. 3:p. 7]. This simplifies the design of the parser because provisions do not have to be made for handling white space and comments. The scanner has already removed them. Also, removing white space and comments and utilizing a fixed-length representation for the tokens saves space. Once tokenization is complete, the source program can be discarded and the compacted tokenized file can be utilized for further analysis.

In order to correctly tokenize the source file there must be some discrete means available to accurately represent each token. There are several ways of describing tokens. One means available is to use a regular grammar. In this method "generative rules are given for producing the desired tokens" [Ref. 3:p. 142]. An equivalent, but different, method is to use finite-state acceptors, FSAs, to recognize tokens. The authors found it easier to visualize this as a recognitive vice generative problem. For this reason the various tokens were modeled using FSAs. An example of an unsigned number recognizer is shown in Figure 2.1. The interested reader is directed to Tremblay and Sorenson [Ref. 3:Chapter 4] for an excellent introduction to the practice of using FSAs to model tokens. The authors found that utilizing FSAs greatly simplified the design, coding, and debugging of the lexical analyzer — one picture was worth a hundred lines of code.

The ideal grammar would allow each token to be uniquely and unambiguously identified. Once the lexical analyzer started on the path of building a token, it would be able to continue until the end with no backtracking. Due to limitations with the standard
Figure 2.1 Unsigned Number Recognizer

ASCII character set, the designer of PHI used multiple keystrokes, or characters, to represent various operators in the language\(^8\). This resulted in compound token types. Also, as in other programming languages, PHI overloads certain operators, allowing them to do double duty\(^9\) by taking on different context-dependent meanings.

The problem of dealing with compound token types was easily handled through the use of a single lookahead character. For example, upon finding the character "-", the scanner looks ahead to the next character to see if it is ">" (\(\rightarrow\)) or another "-" (\(\leftarrow\)). If the next character is neither of these two, it indicates that the token is just the simple token "-". Distinguishing overloaded operators was solved by essentially ignoring it in the scanner! The authors took the position this is basically a syntax analyzer problem and there was no reason to complicate the scanner by handling it. The scanner just identifies a generic token type, e.g. SUB\(_-\), and lets the parser make the proper determination of its true meaning, e.g. SUB\(_-\) or NEG\(_-\).

There are several design decisions relating to the lexical analyzer worth noting. The authors, following the example of Pascal, C, and other languages, took the position that

---

\(^8\)Some examples of this are \(-\rightarrow\) for \(\rightarrow\), \(==\) for \(=\) and \(<>\) for \(\neq\).

\(^9\)For example, + and - can serve as either an unary or binary arithmetic operator.
PHI's keywords\textsuperscript{10} are reserved words and may not be redefined and used as identifiers. Alternate decisions would have been to distinguish keywords from identifiers based on context, as PL/I does, or to precede them by some special character, as ALGOL 60 and ALGOL 68 do [Ref. 3:p. 91]. PHI has a very small set of keywords, smaller than C's, and the authors think that this decision makes life easier for the programmer by simplifying debugging of programs. It certainly made life easier for the authors.

PHI's grammar makes no provisions for programmer comments. The authors originally implemented comments by requiring the programmer to explicitly indicate the beginning and end of each comment with a special character. After scanning the special character at the beginning of the comment, the lexical analyzer would ignore all following characters until the special character was once again found. Following conversations with PHI's designer this implementation was changed. Comments are now implemented the same way they are in Ada\textsuperscript{11}: the comment terminator is the end-of-line character. Not only did this simplify the recognizer for comments, but it also completely removed the problem of runaway comments.

A name table is used to point to the names of all identifiers and constants. A symbol table was originally utilized but later discarded when the authors realized the syntax of PHI makes analyzing an abstract syntax tree easier than analyzing a flattened tree. The information normally associated with a symbol table is now held in the nodes of the tree. This permits just the first instance of each name to be placed into the name table. In other words, regardless of how many times and in how many scopes the identifier X is used, X appears only once in the name table. The token returned to the parser would indicate a

\textsuperscript{10}A complete listing of PHI keywords may be found in the header file for the scanner in Appendix E.

\textsuperscript{11}Ada is a trademark of the Ada Joint Programming Office, Department of Defense, United States Government.
token type of identifier and the parser would then know to dereference the pointer to find the string containing the actual name, X.

Because keywords are reserved, each potential identifier must first be compared against the possible keywords prior to being placed in the name table. The authors implemented a keyword table to simplify this process. Knuth [Ref. 10:pp. 406-410] has shown that a binary search is the most efficient way of searching an ordered table, using only comparisons. For this reason the keyword table is kept in alphabetical order. The lookup, which is at worst \( O(\log n) \), is performed using a binary search of the keyword table.

In an attempt to improve the efficiency of the name table, the authors implemented it as a hash table. McKeeman [Ref. 11:pp. 253-301] experimented with six different length dependent hash functions. He found that the function producing the best results involved summing the internal representation of the first and last characters of the variable's name with its length shifted four places to the left. This was the function utilized by the authors. The possibility of collisions is reduced by choosing a prime number as the table size. However, since this only reduces, not eliminates, the possibility of two or more names hashing to the same value; the authors had to make provisions for handling collisions.

A variant of the chaining method of collision-resolution was chosen. In PHI's implementation, each of the name table slots/buckets holds a data structure that can contain both the name of the variable and a pointer to another structure of the same type. So each hashed value points to a linked list of names. This method offers the advantage of providing better performance than linear probing [Ref. 12:p. 89], is conceptually easy to visualize/work with, and also solves the problem of possibly overflowing the hash table. It does require slightly more memory to implement, but the authors determined that the benefits of this method far outweighed the slight increase in storage requirements. [Ref. 12:pp. 83-93]
B. SYNTACTIC ANALYSIS — THE PARSER

The purpose of the parser is twofold: 1) to determine if the program, as represented by the output from the scanner, is syntactically correct; 2) to impose a hierarchical structure on the token stream, fitting it into the abstract syntax tree which is the output of the parser [Ref. 8:pp. 7–8, Ref. 9:p. 7]. Traditionally, these tasks are done by either a top-down or bottom-up methodology [Ref. 8:p. 41]. Both methodologies use the tokens generated through lexical analysis.

The terminology top–down refers to the order in which the nodes of the parse tree are constructed. Top–down parsing starts from the root of the tree and proceeds downward towards the terminal symbols at the leaves. The parse tree is constructed from the top to the bottom by applying productions of the grammar to generate strings of terminals and nonterminals. On the other hand, bottom–up methodologies start from the terminal symbols at the leaves and proceed upwards to the root. The parse tree is constructed from the bottom to the top by applying reductions of the grammar to generate single nonterminals from strings of terminals and nonterminals. [Ref. 8:pp. 40–41, Ref. 9:pp. 134–136]

It is generally agreed that the popularity of top–down parsing techniques is "due to the fact that efficient parsers can be constructed more easily by hand". [Ref. 8:p. 41] The authors can attest to the fact that the concept of top–down parsing is very easy to grasp. When parsing PHI, it is natural to begin with the start symbol of the grammar, BLOCKBODY, and work forward from there to analyze the token stream. So, partially because of its efficiency, but primarily because of its ease of understanding and use, the authors chose the top–down methodology of recursive–descent parsing to design and implement the syntactic analyzer.

In recursive–descent parsers, separate procedures/functions are written to recognize each nonterminal of the grammar [Ref. 3:pp. 219–220]. This technique gets its distinctive name "because nonterminals can appear in the right–hand sides of each other's
productions, the procedures for recognizing nonterminals are recursive." [Ref.9:p. 150]

To state it more clearly, the function to recognize nonterminal 'A' could end up calling itself recursively if either 1) 'A' appears on the right-hand side of the production describing 'A' itself, or 2) 'A' appears on the right-hand side of the production describing another nonterminal 'B' and 'B' appears on the right-hand side of the production describing 'A'. Regardless of how one looks at the nature of the technique, one usually identifies a stack with recursion. What made this technique so easy to implement was that the authors were able to use C's underlying mechanism for handling recursive functions. The authors did not have to explicitly maintain a stack of symbols for each function call; instead, the information was implicit in the stack of activation records resulting from each function call.

Top-down parsing techniques, especially recursive descent, offer straightforward means of implementing a syntactic analyzer. However, these techniques are applicable only to a subset of the context-free grammars and it is essential that all left recursion be eliminated from the grammar [Ref. 3:p. 211]. In other words, there must not exist any productions describing nonterminal 'A' with 'A' appearing as the first element on the right-hand side of the production. Obviously, if this situation existed, it would be possible to present the parser with strings to parse that would cause it to enter "an infinite loop of production applications" [Ref. 3:p. 211], never to be heard from again. The PHI production QUALEXP = QUALEXP WHERE AUXDEFS is an example of this type of string. The parser would hang up looking for QUALEXP and would never leave this loop until the machine ran out of memory stacking activation records. In order to employ top-down parsing techniques with PHI the authors rewrote the PHI grammar to be right-recursive12. However, as shown below, even the new grammar does not lend itself to "pure" recursive descent parsing techniques.

---

12 The right recursive syntax of PHI may be found in Appendix D
From the compiler writer's point of view, the ideal grammar would allow the correct production rule to be applied in every step of the parsing process. Constructing the parse tree would then proceed in a completely deterministic manner. When this is not possible, there are two basic parser design methods for dealing with nondeterminism in the grammar [Ref. 9: pp. 151–152]. In the backtracking method, which is generally not applicable to recursive-descent techniques, the parser picks an arbitrary production and continues with the parse [Ref. 9: p. 151]. If the parse is successful it is assumed that the correct production was chosen. However, if an error is later discovered, the parser backtracks to the last choice, a new production is chosen, and the parser presses forward again. This process continues until either the parse is successful or the parser runs out of possible productions to choose from. The second method requires a modification to the grammar which results in a deterministic parser: the grammar is rewritten using a process called left factoring to avoid choices among nonterminals [Ref. 9: p. 151].

For the most part, the design of PHI is conducive to recursive descent parsing techniques. There are, however, several productions where this is not so. The result was that a degree of nondeterminism arose in the parser design. The authors attempted to solve this problem through a combination of left factoring and the employment of a simple single token look-ahead. This solution worked for all but the two productions described below. In one case a two token look-ahead was employed and backtracking was used in the other. This is not to say that the authors are absolutely certain that PHI is not an LL(1) grammar or that backtracking had to be used. These solutions were used because they solved the problem at hand.

A two token look-ahead was used for the production\textsuperscript{13} \texttt{ARGBINDING = [QUALEXP \ OP} \)

When the token '[' is found, a flag is set to indicate that an \texttt{ARGBINDING} is being parsed. The first look-ahead token is utilized when parsing the \texttt{QUALEXP} part. \texttt{QUALEXP},

\textsuperscript{13}A complete description of the PHI grammar may be found in the Appendices
for example, may parse as TERM, which in turn may parse as either FACTOR or FACTOR*TERM. After succeeding on FACTOR, a look-ahead is employed to look for the MULOP, *, to see if a recursive search for another TERM should be initiated. This methodology works as long as QUALEXP was not called from ARGBINDING. If it was called from ARGBINDING, argbinding flag set, the operator * could be the trailing operator in the ARGBINDING production and not part of the TERM production. In order to make this determination, an additional look-ahead is utilized to look for the token '\}'. If ']' is found the QUALEXP production is terminated, e.g., term does not recursively call itself again, and the ARGBINDING production is allowed to proceed to completion.

Backtracking was utilized when parsing productions of ACTUAL: ACTUAL = COMPOUND and ACTUAL = DENOTATION = FORMALS $->$ ACTUAL. Legitimate PHI sentential forms produced by the production FORMALS = ( FORMALS+) are proper subsets of the sentential forms produced by the production COMPOUND = ( ELEMENTS ), excluding the empty compound statement. Since any number of identifiers may appear between the parentheses, it is not practical during the parse to utilize look-ahead to determine the presence of the token "$->$". In effect, the parser first realizes it was parsing a DENOTATION when it finds "$->$". This problem was solved by designing the parser to apply first the compound production when presented with this choice. If "$->$" is later found, the parser then backtracks to the FORMALS production. The normal costs associated with backtracking were not evident in this isolated case. As described below, space trade-offs had previously been made and the parser was already working with an abstract syntax tree. The root to the subtree containing the previously parsed compound was simply passed to the FORMALS production to insure that the string could have been

---

14 A purist would say that this instance of backtracking means that the PHI compiler does not in fact employ a recursive-descent parser.
produced by FORMALS. After ascertaining FORMALS, the parser now continues the parse using the DENOTATION production.

The production QUALEXP = QUALEXP WHERE AUXDEFS required a deviation from pure recursive descent parsing. The semantics of this production are such that a terminal (e.g., an identifier) may be used prior to its definition. In itself, this does not present a major problem for the compiler writer. However, this construct also changes the scope of the identifier since the inner-most scope, in the form of the QUALEXP, is parsed first and the parser then works its way to the outer-most scopes, the AUXDEFS. This problem is analogous to that of mutual recursion in Pascal, without the benefit of the forward declaration [Ref. 4:p. 213].

Originally, the parser was designed to output the parse tree in flattened form, essentially a post-order walk of the tree. This design implemented traditional symbol-table management routines. However, after obtaining a clearer understanding of the semantics involved with the problems mentioned earlier, notably the production QUALEXP = QUALEXP WHERE AUXDEFS, the authors realized a traditional symbol-table would be too inefficient. Management of the table would take an inordinate amount of assets and be too unwieldy to work with. The authors solved this problem by maintaining the status of the parse in an abstract syntax tree so the output from the parser is now in tree form. This permits information originally held in the symbol-table to be maintained in the tree itself. The parser is able to analyze the source program by walking the tree and decorating the nodes with required information. Maintaining a binary tree in memory does require more space, but this is insignificant when compared with the benefits.

Interestingly, maintaining the parse in tree form presented several additional benefits. The solution to the aforementioned problem of distinguishing between COMPOUND and DENOTATION became trivial because it was now simply a matter of returning to the appropriate subroot and rewalking the tree. Also, working with a binary tree permitted the
authors to perform a modicum of optimization in the parser. It becomes relatively straightforward to perform compaction on an actual tree.

The authors think that this design offers maximum potential for future enhancements of the PHI programming environment. One possibility would be to use this front-end to drive a PHI interpreter. Modularization of the front-end in this manner simplifies functional understanding of the front-end and should lead to increased ease of maintenance and portability. To demonstrate portability, the authors recompiled the front-end and executed it on a 68000 based processor. This was accomplished with no modifications to the source program, just replacement of C run-time header files for the new target machine.

C. ERROR HANDLING

Tremblay and Sorenson [Ref. 3:p. 183] classify error responses into three categories:

I. Unacceptable responses
   1. Incorrect responses (error not reported)
      a. Compiler crashes
      b. Compiler loops indefinitely
      c. Compiler continues, producing incorrect object program
   2. Correct (but nearly useless)
      a. Compiler reports first error and then halts

II. Acceptable responses
   1. Possible responses
      a. Compiler reports error and recovers, continuing to find later errors if they exist
      b. Compiler reports the error and repairs it, continuing the translation and producing a valid object program
   2. Impossible with current techniques
      a. Compiler corrects error and produces an object program which is the translation of what the programmer intended to write

In the prototype PHI compiler, the authors have implemented a limited form of error recovery. The primary benefit of error recovery is to "prolong the compilation life of the program as long as possible before the compiler gives up on the source program". [Ref. 3:p. 11] This allows the maximum number errors to be discovered per compilation, shortening the edit, compile, debug cycle inherent to writing computer programs.

The authors analyzed the intended environment and use of the PHI compiler and decided that lexical analysis and syntactic analysis were the most likely source of errors.
Lexical errors basically involve invalid characters or incorrect tokens. Common examples of these types of errors are unrecognized words, misspelled identifiers/keywords, or illegal characters. Syntactic errors relate to incorrect structure of the program. These errors arise when the programmer failed to follow the rules, productions, of the grammar. The form of the program is wrong. [Ref. 9:p. 226, Ref. 3:p. 185]

One thing the error handler should not do is exacerbate the situation by reporting bogus errors or executing an erroneous program. To insure erroneous programs are not executed, the authors inhibited object file production if any errors were discovered. The authors do not believe the compiler should allow code generation to continue, or even begin, if the source program has errors. Often times one error leads to an avalanche of errors being reported and this is extremely annoying to the programmer. The authors attempted to minimize this situation, but found it impossible to eliminate completely because some errors feed on others. To insure the programmer would not become overwhelmed with error messages, the authors terminate the compilation after 10 errors. Also, for programmer convenience, actual error messages are outputted instead of error codes. The authors saw no justification in using a cryptic code when a plain language message served much better. Since the authors anticipate students in functional programming classes to be primary users of the PHI compiler, error messages have their basis in the productions describing the PHI language. It is assumed that users of the PHI compiler have an understanding of PHI's syntax.
III. BACK-END OF THE COMPILER

A. OVERVIEW

The back-end of the compiler consists of the semantic checker and code generator. Semantic checking and code generation are completed in one pass, and the output is a sequence of bytes, held in memory, which correspond to ASCII characters. These characters are then written to a text file, which the assembler uses to output an object file. This output is linked to the appropriate run-time routines to make a usable program. For the current implementation, a RASM86 assembler and LINK86\textsuperscript{15} linker are used.

B. RUN-TIME ORGANIZATION

Since PHI is a structured language with scoping and function calls, it lends itself to a stack-oriented run-time architecture. The stack is set up to accomplish two tasks: 1) to hold pointers to the current operands, and 2) to hold activation records for functions currently in use. Both of these tasks are described below.

There is a 64 kilobytes limit on memory used while a program is running. This limitation is imposed because the memory is addressed as an offset from a base address, and the maximum offset is 64K. This space is competed for by the stack, current variables, and constants (see Figure 3.1). The stack grows from the top of this space down, and the variable space grows from the base of this space up, preventing wastage by either component. Because PHI is a functional language, a value is returned from each operation, and a pointer to this value is placed at the top of the stack. The returned value is placed in the lowest available space in the part of memory assigned to variables and constants. A heap allocation method is not currently used because 1) all data types currently implemented use only one word of memory, and 2) there is no fragmentation of

\textsuperscript{15}RASM86 and LINK86 are trademarks of Digital Research, Inc.
memory because all types are currently static. If the next operation is a binary operation, a pointer to the second operand is placed on the stack, and the operation takes place using the two topmost pointers. The result is placed in memory, and the process begins afresh with new operands. If the next operation is unary (such as the negation operation), no change to the stack takes place and the variable in memory is altered as the program directs.

![Figure 3.1](#)

**Figure 3.1**
Memory Organization

If the second operand of an operation is to be the result of a function call (e.g., "2 * f(x)"), an activation record is placed on top of the pointer to the first operand and the function's value is calculated. Then, the activation record is deleted and a pointer to the function result is saved and placed at the top of the stack.
The activation record itself, Figure 3.2, contains three parts: the static link, the static nesting level, and a pointer to the address in memory where the function's first variable is stored. The static link is a one-word pointer which points to the static nesting level space of the previous activation record, and is used to traverse the stack from activation record to activation record, i.e. a static chain. [Ref. 4: p. 77]. The static nesting level and the pointer to the base of the storage space for a scope's values are used to access variables and constants. In this design, a two-tuple \((B, L)\) is associated with each variable. In this two-tuple, \(B\) represents the static nesting level and \(L\) is the offset within that level. By following the static chain for \((\text{current nesting level} - \text{target nesting level})\) links, the activation record of the scope of the target value can be accessed. Then, the address of the variable is calculated by adding \(L\) to the low address of the scope's variables. An alternate method would have been to store the values directly in the stack between or within activation records. However, this is a messy process when dealing with dynamic data structures such as sequences. Additionally, it is conceptually easier to divide the stack and the variables.

Functions are implemented as calls to assembly language subroutines, with pointers to the arguments placed on the stack before calling the routine. Using this scheme and noting the fact that PHI cannot have side effects, the implementation of recursion is straightforward. Whenever a function is called, its activation record is placed on the stack and pointers to its arguments are placed on top of the activation record. If the function is
recursive, the assembly language subroutine simply calls itself until the base of its
recursion is reached or until stack overflow is reached. Figure 3.3 shows an example of a
series of activation records called by a program with a recursive function. Note that the
data definition ("answer") has no arguments and simply calls the factorial function. The
factorial function, on the other hand, has an argument and it uses that argument as an
operand. So, a pointer to that value is put on the stack and the next operand, fac (n - 1), is
put on the stack as an activation record. When fac (n - 1) is evaluated, a pointer to its
return value is placed on the stack. This cycle of evaluation, pop activation record,
evaluation will continue until the data definition "answer" is evaluated.

```
answer where answer == fac(5) where
fac(n) == if n == 0 then 1
else n * fac(n-1) endif
```

Figure 3.3
Factorial Program and Activation Records

As an example of the code generated for function calls and recursion, the following
PHI program fragment is used:  
"C (n) == if n = 0 then 1 else C (n - 1) * n endif."

28
This, of course, simply calculates the factorial of the integer n. Figure 3.4 is the listing of the assembly language segment which is generated from this fragment.

<table>
<thead>
<tr>
<th>Address/Machine Code</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>0103 E94A00</td>
<td>0150 jmp a10000</td>
</tr>
<tr>
<td></td>
<td>a10001:</td>
</tr>
<tr>
<td>0106 B90000</td>
<td>mov cx,0</td>
</tr>
<tr>
<td>0109 E80000</td>
<td>call i_formal</td>
</tr>
<tr>
<td>010C B80000</td>
<td>mov ax,0</td>
</tr>
<tr>
<td>010F E80000</td>
<td>call iequ</td>
</tr>
<tr>
<td>0112 E80000</td>
<td>call igrvalue</td>
</tr>
<tr>
<td>0115 E80000</td>
<td>cmp ax,1</td>
</tr>
<tr>
<td>0118 3D0100</td>
<td></td>
</tr>
<tr>
<td>011B 7509</td>
<td>0126 jne a10003</td>
</tr>
<tr>
<td>011D B80100</td>
<td>mov ax,1</td>
</tr>
<tr>
<td>0120 E80000</td>
<td>call iputvalue</td>
</tr>
<tr>
<td>0123 E92600</td>
<td>014C jmp a10002</td>
</tr>
<tr>
<td></td>
<td>a10003:</td>
</tr>
<tr>
<td>0126 B90000</td>
<td>mov cx,0</td>
</tr>
<tr>
<td>0129 E80000</td>
<td>call i_formal</td>
</tr>
<tr>
<td>012C B90000</td>
<td>mov cx,0</td>
</tr>
<tr>
<td>012F E80000</td>
<td>call i_formal</td>
</tr>
<tr>
<td>0132 B80100</td>
<td>mov ax,1</td>
</tr>
<tr>
<td>0135 E80000</td>
<td>call iputvalue</td>
</tr>
<tr>
<td>0138 E80000</td>
<td>call isub</td>
</tr>
<tr>
<td>013B E80000</td>
<td>call ppop</td>
</tr>
<tr>
<td>013E 51</td>
<td>push cx</td>
</tr>
<tr>
<td>013F 57</td>
<td>push di</td>
</tr>
<tr>
<td>0140 BB0100</td>
<td>mov bx, 1</td>
</tr>
<tr>
<td>0143 E80000</td>
<td>call i_mov</td>
</tr>
<tr>
<td>0146 E8BDFF</td>
<td>0106 call a10001</td>
</tr>
<tr>
<td>0149 E80000</td>
<td>call imult</td>
</tr>
<tr>
<td></td>
<td>a10002:</td>
</tr>
<tr>
<td>014C E80000</td>
<td>call del_scope</td>
</tr>
<tr>
<td>014F C3</td>
<td>ret a10000:</td>
</tr>
</tbody>
</table>

Figure 3.4
Assembly Language Output from Factoral Program

The label "a10001" at address 0103 is the label of the subroutine which returns the factorial. When it is called, pointers to the values of the arguments are placed on the stack. If the subroutine is called before the base of the recursion is reached, a jump is made to label a10003. Then, the new actual value (n - 1) is calculated and placed in the low part of memory, a pointer to the value is put on the stack, and the values are prepared for calling
by the next subroutine (lines 0126 to 0143). The factorial subroutine is then called again. This process continues until the base of the recursion is reached; in this case a pointer to the integer value is put at the top of the stack (line 011D), and a jump is made to label a10002. Here, the subroutine "del_scope" tears down the activation record on the stack and puts a pointer to the result of the function at the top of the stack. Clearly, recursion in the PHI program can be implemented by a parallel recursion in the assembly language output of the compiler.

Another feature of the output code shown in Figure 3.4 is that there is an unconditional jump around the function (lines 0103 and 014F). This is a result of the decision to output inline code in spite of the fact that functions can be called at random. There are both space and time penalties to be paid for these jumps, especially since each function must have a jump and label instruction bracketing it. However, the ultimate effect of all these jumps is to get to the label at the bottom of the program. The result is that all but one jump/label pair could be eliminated by an optimizer, making the penalty trivial. Another solution considered was to generate code for functions and the "main" program separately, then combine the two when printing the output from the code generator. This was not done for reasons put forth in the section that describes the semantic analyzer.

Variable and constant storage is word oriented rather than byte oriented to take advantage of the 8086 processor's 16 bit capability. Integers and naturals are both represented as single words, and booleans are represented as integers, either 1 or 0. While this boolean representation is somewhat wasteful in terms of memory space, it allows for a great deal of overlapping in certain subroutines used in function calling and comparisons. It is planned to represent real numbers with two words of memory, and sequences using linked lists. Neither of these types have been fully implemented; however, there are provisions in the compiler for adding these features at a later date.
There is currently no dynamic allocation of registers. Some registers are used for specific purposes; for instance, the SI register is used to mark the top of the program stack, and of course the BP and SP registers are used to manage the machine's stack. In general, arithmetic processes take place in the AX register, using other general registers as auxiliaries as needed. When variable space is needed, the highest unused address space is allocated and, when a function is finished, only the result is saved in storage; all other value spaces are returned for use by the program.

Error handling is probably the simplest part of the run-time routines. Any run time error such as overflow or division by zero errors will result in an appropriate error message to the user (see Appendix O for a full listing of error messages). Then, program execution will terminate and control is returned to the operating system.

C. SEMANTIC CHECKING and CODE GENERATION

The PHI compiler utilizes the recursive descent technique to perform semantic checking and code generation in one traversal of the parser tree. In most cases, tree nodes are filtered through the semcheck function, which calls various procedures based on the name of the node. These procedures, in turn, call semcheck for each of their children, and the process is repeated until the leaves of the tree are reached. The function semcheck then returns a type (e.g., integer, real, boolean), which the parent node uses to determine the semantic correctness of its subtree. With the information returned from the semcheck function, the parent procedure can do one of three things: return a type, convert one node to a different type, or declare an error condition.

Concurrent with semantic checking, code is generated. As noted above, this is assembly language code written to a buffer in memory. If an error condition is declared, however, a flag is set and code generation ends. Semantic checking will then continue until the tree is completely traversed or ten errors are accumulated; then, the semantic checking
process terminates. Unlike the parser, the semantic checker makes no attempt at error recovery; top-down checking simply continues normally from where the error was detected.

Top-down semantic checking results in a neat, trim package for the back end of the compiler. Unfortunately, there are some problems that pure top-down checking will not solve. For instance, determining if there is a one-to-one match between formals and actuals for a given function involves some detours from top-down checking, as explained below.

The scoping rules of PHI provided the largest challenge to writing the semantic checker. One solution is a multiplicity of stacks. The size of these stacks depends upon the number of its constituents visible at any one time. Usually, the proper match for an item is the one found closest to the top of the stack. However, because of the semantics of the "and" construct, checks against the variable-stack do not always follow this convention.

There are four stacks used by the semantic checker: the type-stack, the variable-stack, the definition-stack, and the and-stack. All but the type-stack are implemented as linked lists. This implementation sheds the disadvantage of static length arrays at the cost of a slight increase in memory and temporal resources. The type-stack uses a fixed-length array of 300 entries because 1) the basic types of real, boolean, integer, natural, and trivial will be accessed most frequently, because they are the building blocks of every type and sequence, and because they can be more easily accessed from an array than from a linked list, 2) a list of 300 type entries should not impose an extreme burden on the programmer, and 3) the planned implementation of sequences will be more straightforward if the type-stack is an array.
The type-stack, Figure 3.5, is meant to hold both the basic type definitions and user defined type definitions. This stack holds both the name of the type and the number of bytes needed in memory to implement the type. At compiler initialization, it contains the five basic types and user defined types are added as they are encountered. The begin–end construct of the language (not implemented yet) allows declared types to be visible over a specified range. It is planned to implement this construct by setting a pointer to the top of the stack upon encountering the begin node and then popping the stack to that point after both of the node's subtrees have been checked.

The variable-stack, Figure 3.6, holds all of the variables, including function names, currently seen by the semantic checker. Each entry holds a pointer to the hash table containing labels, a type, a pointer to the tree node defining it, and a flag to designate whether or not it is a formal. Whenever a variable name is encountered and the name is not a call to a function and not a data definition, it is put into the variable stack. Then, when a scope is exited, the variables local to that scope are dropped from the stack. For example, after a function is defined, all of its formals are popped from the stack.
The definitions–stack, Figure 3.7, contains all of the function and variable definitions visible in a given scope; e.g., the declaration $C : \star \mathbb{R} \rightarrow \mathbb{B}$ would put the definition $C$ into the definition–stack. This entry would contain the type of $C$'s return value (Boolean), a pointer to the tree node that contains $C$, and a pointer to a linked list which contains its argument types (Real and Integer). This last field will be null if the declaration is a data definition. This stack grows and shrinks in the same way as the type stack.

The authors considered combining the definitions–stack and the variable–stack because of the similarity between their fields. In fact, one of the primitive implementations was designed in this way. However, this slowed down the search for both definitions and variables considerably, and the overhead needed to implement these two as separate stacks is small: three extra functions and one extra pointer.

The need for the and–stack is derived from the scoping rules imposed by the AND construct. This construct allows a variable to be referenced before it is declared without the benefit of Pascal's forward declaration or equivalent. This is true of other constructs in PHI such as the WHERE construct. However, the AND construct cannot be parsed in such a way that the semantic checker can see all variables before they are used, because either subtree of the AND statement can define variables used by the other subtree. So, a program such as the one depicted in Figure 3.8 needs a vehicle by which it can detect that the variable $d$ is defined later in the program. The and–stack is such a vehicle.
When the semantic checker reaches the AUXAND node, Figure 3.8, a flag is set to indicate that AUXAND has been traversed, and a pointer is set to the top entry of the and-stack. "Notfound" is returned from the semcheck function when the variable d is reached, but, since the AND condition has been set, a pointer to d is put in the and-stack. Note that d is later defined in a data definition (DATAUXDEF node), and when both the left and right subtrees of AUXAND have been checked, all variables in the and-stack are checked against variables in the variable-stack. If a match is found, d is defined and removed from the and-stack. In the event that a variable is not found when the AUXAND node's complete subtree has been checked, an error condition (UNDEFINED VARIABLE) would be set. The semantic checker would recognize this condition because the top of the and-stack would not be equal to the mark placed at the top of the stack when the AUXAND node was entered. Nested AUXANDS are possible, but they pose no problem because the top of the and-stack is marked when the auxand node is traversed.
Variables and functions are represented in the run-time by a call to an assembly language subroutine, and each subroutine must have a discrete name. Also, there are several labels found throughout the program, and each of these must have a name. These names are generated by the "name" function found in the sem_u.c module. Each name begins with the letter "a", followed by 6 digits. Examples can be seen in Figure 3.4.

\[ f(x,y) = x \times y \]

Figure 3.9
Tree for Function f

Function definitions presented a problem that was solved with a deviation from pure top-down semantic checking. When a function definition (FUNAUXDEF in Figure 3.9) is encountered by semantic checker, the following procedure would be followed (see Figure 3.10 for the function definition entry):

```
funid_node:
    check for definition-stack entry for "f"
    if not found
        return (ERROR)
    get a pointer to the first formal of f
    get a pointer to the first formal of definitions-stack entry
    while both pointers <=> Nil do
```

36
put variable in varstack; use type pointed to by the formal list
advance both pointers
end while loop
if not (both pointers == nil)
return (FORMALS MISMATCH)
else
put "f" in the variable-stack
return (Type of f = INTEGER)
end else
end.

**funauxdef node:**

left type = semcheck (Left Child)
right type = semcheck (Right Child)
if (left type <> right type)
call a procedure which will either
convert the right type to the left type or set an error flag.
endif
end.

When a function is called with arguments, a similar process takes place (refer to Figure 3.11):

**actualist :** Input is a pointer to the actualist node
Output is error condition
Check definitions-stack for "f"
if "f" not found
set error (FUNCTION DEFINITION NOT FOUND)
set elistptr to first element of elist list
elist (elistptr)
check var stack for "f"
if found,
generate code to call "f"
if not found
if and_flag = TRUE
put "f" in the and stack
else
set error (FUNCTION NOT DEFINED)
end.

**elist :** Input is a pointer to the element list node
if pointer->rptr <> nil
elist (pointer->rptr)
check type of element against corresponding formal type
if types don't match
set error (IMPROPER ARGUMENT TYPE)
else
    generate code to put pointers to argument values on the run-time stack
end.

![Diagram](image_url)

**Figure 3.10**
Definitions—Table Entry For Function f

Type conversions are implemented in the semantic checker, albeit the code generator does not yet support this feature. The function `hnumconvert` (half number-convert, found in the module `sem0`) will check to see if a conversion of the right subtree of a node to the left subtree type should be accomplished. This is useful for function definitions, where the body of the function may be converted to the type the function returns, but the converse is not acceptable. In addition, the function `numconvert` (found in the `sem0` module) will convert either the left tree type or the right tree type of a node. This is useful for certain arithmetic operations. The semantic checker considers integer-to-real and natural-to-real conversions to be legal. Natural to integer conversions are not implicitly done, since both of these types are represented in exactly the same way. On the other hand, an attempt to return an integer value for a function which has a declared type of natural will result in an error.
Variables of simple type (i.e., natural, integer, or real) need not be declared before use, although such a declaration may be made. If a variable is undeclared when defined by a data definition, the semantic checker will attempt to classify it. If the semantic checker expects to find a boolean value, the variable is easily classified as a boolean and an entry is put into the variable table. If a numeric variable is expected, the semantic checker will try to type it as an integer; failing this, it will be classified as a real number. However, the AND construct alters this somewhat. If a variable is used before it is defined by a data definition, it must have been defined using the LETDEF construct.

As noted in the section on run-time, some thought was given to generating all functions and data definitions to one buffer and the "main" program which calls these functions to another buffer. However, this would be an inefficient use of memory space.
since one buffer might run out of space while the other is under-utilized. Although there is a proliferation of jump calls in the output using one buffer, an optimizer could easily eliminate all but one call, as noted above.

D. OPTIMIZATION

There is no optimization module implemented in the PHI compiler. In this section an attempt will be made to identify three types of optimization which are suitable for implementation. Also, a small dissertation on what optimizations should not be considered is included.

The first suitable type of optimization is constant folding. The purpose of constant folding is to eliminate multiple consecutive constants in arithmetic expressions [Ref 3:p. 612], and the function numconvert in module sem0 makes an excellent structure in which to implement this optimization. This is because most arithmetic operations call this function. It would be straightforward to put a function that tests the left and right children of an operand node to see if they are constants, then perform the operation in the compiler and generate code for a constant call. However, since the division operators do not call numconvert, the constant folding function would have to be inserted in idiv and rdiv also.

The other two optimizations are post-code generation optimizations. The first one considered is jump optimization. This should be the most worthwhile to implement: if the number of functions and data definitions is n, n > 0, there will be n - 1 unnecessary unconditional jump statements and labels.

These jump statements can be eliminated by replacing the first "jmp" statement with a jump to the last label in the code; then, because "jmp" is not used for anything except to circumnavigate functions and data definitions, all other unconditional jumps and their labels can be eliminated.
The last type of optimization is a form of peephole optimization. Occasionally, there will be a "call push" statement followed by a "call pop" statement. This is unnecessary, and can be eliminated. The 8086 assembly code equivalent of push followed by pop should not occur in the present design.

Dead code optimization eliminates code inside a jump when that code contains no labels. It is not necessary to implement this type of optimization with the current design since unconditional jumps are only used to bracket functions and definitions. However, if one accepts the premise that programmers occasionally make mistakes, it might be worthwhile to keep track of which functions are called and eliminate code for those which are not. A message to the programmer concerning this circumstance would be useful too.
IV. RESULTS & CONCLUSIONS

A. RESULTS

The implementation described in this study demonstrates the design and implementation of a compiler for the functional programming language PHI. Since this implementation is a prototype, it does not possess all of the qualities desirable in a full implementation. However, the necessary hooks are present and the design is mature enough to allow expanding the prototype to a full implementation.

The PHI compiler front-end implements machine independent lexical and syntactic analyzers. This implementation is complete and faithfully follows the syntax of PHI — based on the design of the language as of 07 January 1987. In deciding which modules to include in the front-end and back-end, the authors were originally guided by the traditional methodology of placing the analysis functions in the front-end and generative functions in the back-end (Ref. 8, p. 201). However, as the design of the PHI compiler progressed, the authors removed semantic analysis from the front-end and combined it with code generation. This produced a one-pass semantic analysis/code generation phase.

The PHI compiler back-end implements a machine dependent one-pass semantic analyzer and Intel 8086 code generator. The semantic analyzer implements the basic constructs of PHI: functions and data definitions may be defined, and the integer, natural number, real number, and boolean types are fully implemented. Implementation of code generation is congruent to that of the semantic analyzer, with the exception that the real number data type has not been implemented.
B. CONCLUSIONS

It is possible, using traditional technologies to design and implement a compiler for the functional programming language PHI. It is not possible to utilize either pure recursive descent or pure deterministic techniques for this implementation. The syntax/semantics of the language forced a degree of non-determinism, and one instance of back-tracking was required in the PHI compiler front-end.

The overall design is highly modularized facilitating the understanding of concept and implementation. The authors think that this approach will greatly reduce maintenance costs and provide greater flexibility in making changes and additions to the PHI programming environment. It should be possible, for example, to use the front-end described in this thesis to drive a PHI interpreter. Being able to abstract out this front-end and use it without change should make the implementation of a PHI interpreter relatively simple. Modularizing the design also increases portability of the compiler to other machines. To demonstrate portability, the authors recompiled the front-end and executed it on a 68000 based processor. This was accomplished with no modifications to the source program, just replacement of C run-time header files for the new target machine.

Removing the semantic analyzer from the front-end permitted coupling semantic analysis with code generation. The fixed-length buffer design of the code generator is suitable for this prototype implementation but should be redesigned utilizing dynamic buffer allocation methods in follow on implementations. The authors think that utilizing a single pass through the parse tree is practical for the basic constructs of PHI and believe this methodology is suitable for future designs of the PHI compiler.
V. FURTHER RESEARCH

Further research may be broken down into two major areas: short and long range projects. The former may be further broken down into two main areas: adding unimplemented features and improving the PHI programming environment. On the other hand, all long-range projects involve only the programming environment. All of these areas are discussed below.

In the prototype of the PHI compiler, both Real and Compound variable types remain unimplemented. Compound variable types consist of sequences, the Trivial type, user defined types, and tuples. Although all of these are recognized by the parser, the semantic checker will not recognize complex types and no code will be generated. The Real type is recognized by the semantic checker, which can discern if conversion from an integer or natural type should be accomplished; however, no code is generated to implement this type in the run-time structures. Note also that operators which operate solely on complex types and reals (e.g., the real divide and concatenate operators) are not implemented.

One other operator not implemented is the "|->" operator. In addition, argument bindings, functionals, and FILEs are not recognized by either the semantic checker or the code generator.

Short-range improvements to the PHI environment may come either after a full implementation is accomplished or may be developed concurrently with the full implementation. Admittedly, the current environment is analogous to instrumentation on a helicopter: there is just enough to know that the system is running! The environment could be improved by implementing the interactive mode of PHI, as opposed to the current batch mode. A sample interactive session of PHI may be found in [Ref. 1:pp 1-17]. Also, an interpreter would be a good starting point toward developing a practical, working
environment for PHI. As noted above, the front end of the prototype compiler may be
adapted for this purpose; alternatively, due to the structural similarities between PHI and
LISP, an ambitious researcher may wish to write an interpreter in LISP.

One final short-range improvement which is not covered by either category would be
to allow more than 64K of run-time memory. It would be worthwhile to take advantage
of the large amount of memory most modern microcomputers have, especially since
sequences and recursion, upon which PHI is based, gobbles up memory with abandon.

When the PHI compiler becomes a serious user's tool, some long-range research will
become viable. Sophisticated input and output would be a vital consideration, and the
minimal I/O methods now in use would need substantial improvement. The most
ambitious researchers in this direction should consider a bit-mapped display with the
possibility of a syntax-directed editor. Also, based on the authors' limited experience in
PHI programming, a debugger would be a necessary tool for the serious programmer.
LIST OF REFERENCES


APPENDIX A
THE FUNCTIONAL LANGUAGE PHI — Φ
(CONCRETE SYNTAX OF Φ — 10/16/86)

Grammatical Notation:

Both ‘{C_1,C_2,...,C_n}’ and \[\begin{array}{c} C_1 \\ \vdots \\ C_n \end{array}\] mean exactly one of \(C_1, C_2, ..., C_n\).

Similarly, ‘[C_1 | ... | C_n]’ and \[\begin{array}{c} C_1 \\ \vdots \\ C_n \end{array}\] mean at most one of \(C_1, ..., C_n\). The notation ‘C*’ means zero or more Cs; ‘C^+’ means one or more Cs; ‘CD ...’ means a list of one or more Cs separated by Ds. Terminal symbols are quoted when they could be confused with metasymbols.

Grammar:

\[
\begin{align*}
\text{BLOCKBODY} & = \{ \text{QUALEXP} \\
& \quad \{ \text{LET DEFS ; BLOCKBODY} \} \} \\
\text{DEF} & = \{ \{ \text{ID} \} \text{FORMALS = QUALEXP} \\
& \quad \{ \text{ID} : \text{TYPEEXP} \\
& \quad \{ \text{TYPE ID [FORMALS] = TYPEEXP} \} \} \} \\
\text{QUALEXP} & = \{ \text{EXPRESSION WHERE AUXDEFS} \\
& \quad \text{QUALEXP WHERE AUXDEFS} \} \\
\text{AUXDEFS} & = \text{AUXDEF AND ...} \\
\text{AUXDEF} & = \{ \text{ID} \} \text{FORMALS = EXPRESSION} \\
\text{FORMALS} & = \{ \text{ID} \\
& \quad \{ \text{FORMALS, ...} \} \} \\
\text{EXPRESSION} & = \{ \text{EXPRESSION \lor \text{CONJUNCTION} \} \\
\text{CONJUNCTION} & = \{ \text{CONJUNCTION \land \text{NEGATION} \} \}
\end{align*}
\]
NEGATION = [ ~ | RELATION
RELATION = [SIMPLEXP RELATOR] SIMPLEXP
RELATOR = ( = | ≠ | > | < | ≤ | ≥ | ∈ | ∉ )
SIMPLEXP = [SIMPLEXEP ADDOP] TERM
ADDOP = ( + | - | : | ^ )
TERM = [TERM MULOP] FACTOR
MULOP = ( ✕ | / | + )
FACTOR = [ + ] primary

PRIMARY = { APPLICATION
            PRIMARY APPLICATION }
APPLICATION = [APPLICATION] ACTUAL

ACTUAL = { ID
          DENOTATION
          CONDITIONAL
          COMPOUND
          ARGBINDING
          BLOCK
          FILE ' CHAR+' }

DENOTATION = { ' CHAR*' 
               DIGIT* [. DIGIT* ] 
               FORMALS → ACTUAL }

CONDITIONAL = IF ARM ELSIF ... [ELSE EXPRESSION] ENDIF
ARM = EXPRESSION THEN EXPRESSION

COMPOUND = { ( ELEMENTS ) 
             ( ' ELEMENTS ' ) 
             < ELEMENTS > }

ELEMENTS = [QUALEXP, ...]

ARGBINDING = ( OP OP QUALEXP , QUALEXP OP ) ' |
OP = ( , | RELATOR | ADDOP | MULOP | ! )

BLOCK = BEGIN BLOCKBODY END
DEFS = DEF AND ...

TYPEEXP = TYPEDOM [→ TYPEEXP ]

TYPEDOM = TYPETERM [+ TYPEDOM ]

TYPETERM = TYPEFAC [ X TYPETERM ]

TYPEFAC = { TYPEPRIMARY
            TYPEPRIMARY* }
            ID << TYPEEXP, ... >>

TYPEPRIMARY = { ID
               PRIMTYPE
              ( TYPEEXP ) }

PRIMTYPE = ( R | Z | N | B | 1 | TYPE )

For batch use, a program is considered a BLOCKBODY; for interactive use it is considered a SESSION:

SESSION = COMMAND+

COMMAND = { DEF
            QUALEXP } ;

49
APPENDIX B
THE FUNCTIONAL LANGUAGE PHI — Φ
(CONCRETE SYNTAX OF Φ — 03/03/87)

Grammatical Notation:

Both \( \{C_1, C_2, \ldots, C_n\} \) and \( \left\{ \begin{array}{c} C_1 \\ C_2 \\ \vdots \\ C_n \end{array} \right\} \) mean exactly one of \( C_1, C_2, \ldots, C_n \).

Similarly, \( \{C_1 \mid \ldots \mid C_n\} \) and \( \left[ \begin{array}{c} C_1 \\ \vdots \\ C_n \end{array} \right] \) mean at most one of \( C_1, \ldots, C_n \). The notation \( C^* \) means zero or more Cs; \( C^+ \) means one or more Cs; \( CD \ldots \) means a list of one or more Cs separated by Ds. Terminal symbols are quoted when they could be confused with metasymbols.

Grammar:

\[
\begin{align*}
\text{BLOCKBODY} & = \{ \text{QUALEXP} \\
& \quad \text{LET DEFS ; BLOCKBODY} \} \\
\text{DEF} & = \{ \text{REC} \left( \begin{array}{c} \text{[ID, \ldots : TYPEEXP BE IS ]} \\ \text{[ID] FORMALS = QUALEXP} \\
& \quad \text{TYPE ID [FORMALS] = TYPEEXP} \end{array} \right) \} \\
\text{QUALEXP} & = \{ \text{EXPRESSION} \\
& \quad \text{QUALEXP WHERE AUXDEFS} \} \\
\text{AUXDEFS} & = \text{AUXDEF AND} \ldots \\
\text{AUXDEF} & = \text{[ID] FORMALS = EXPRESSION} \\
\text{FORMALS} & = \{ \text{ID} \\
& \quad \text{([FORMALS, \ldots])} \} \\
\text{EXPRESSION} & = \{ \text{EXPRESSION V} \} \text{CONJUNCTION} \\
\text{CONJUNCTION} & = \{ \text{CONJUNCTION A} \} \text{NEGATION}
\end{align*}
\]
NEGATION = \[ \sim \] RELATION
RELATION = \[ \text{SIMPLEXP RELATOR} \text{SIMPLEXP} \]
RELATOR = \{ = | \neq | > | < | \leq | \geq | \in | \notin | \rightarrow \}
SIMPLEXP = \[ \text{SIMPLEXP ADDOP} \text{TERM} \]
ADDOP = \{ + | - | : | ^ | | \}
TERM = \[ \text{TERM MULOP} \text{FACTOR} \]
MULOP = \{ x | / | \# \| \ltt \mid X \}
FACTOR = \[ \text{+} \mid \_ \] PRIMARY
PRIMARY = \{ APPLICATION PRIMARY APPLICATION \}
APPLICATION = \[ \text{APPLICATION} \text{ACTUAL} \]
ACTUAL = \{ ID \{ \text{TYPEEXP} \mid \ldots \} \DENOTATION \}
CONDITIONAL = IF \text{ARM} ELSIF \ldots [\text{ELSE EXPRESSION}] \text{ENDIF}
ARM = \text{EXPRESSION \text{THEN} EXPRESSION}
COMPOUND = \{ (' \text{ELEMENTS} ') \}
\{ ( \text{ELEMENTS} ) \}
( \text{\' ELEMENTS \'} )
< \text{ELEMENTS} >
ELEMENTS = \[ \text{EXPRESSION} \mid \ldots \]
ARGBINDING = \{ \text{OP \text{ACTUAL \text{OP}}} \}
OP = \{ , | \text{RELATOR} | \text{ADDOP} | \text{MULOP} | \text{SUB} \}

51
\begin{align*}
\text{BLOCK} &= \text{BEGIN BLOCKBODY END} \\
\text{DEFS} &= \text{DEF AND ...} \\
\text{TYPEEXP} &= \text{TYPEDOM} [ \rightarrow \text{TYPEEXP} ] \\
\text{TYPEDOM} &= \text{TYPETERM} [ + \text{TYPEDOM} ] \\
\text{TYPETERM} &= \text{TYPEFAC} [ \times \text{TYPETERM} ] \\
\text{TYPEFAC} &= \left\{ \begin{array}{l}
\text{TYPEPRIMARY}\ast \\
\text{TYPEPRIMARY} [ \text{ACTUAL} ]
\end{array} \right\} \\
\text{TYPEPRIMARY} &= \left\{ \begin{array}{l}
\text{ID} [ \ast \text{TYPEEXP}, \ldots ] \\
\text{PRIMTYPE} \\
(\text{TYPEEXP})
\end{array} \right\} \\
\text{PRIMTYPE} &= [ \text{R} | \text{Z} | \text{N} | \text{B} | 1 | \text{TYPE} ]
\end{align*}

For batch use, a program is considered a BLOCKBODY; for interactive use it is considered a SESSION:

\begin{align*}
\text{SESSION} &= \text{COMMAND}\ast \\
\text{COMMAND} &= \left\{ \text{LET DEF} \right\};
\end{align*}
## APPENDIX C

**ASCII REPRESENTATION OF — Φ**

<table>
<thead>
<tr>
<th>Reference</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>==</td>
</tr>
<tr>
<td>&lt;</td>
<td>LESS</td>
</tr>
<tr>
<td>≤</td>
<td>&lt;=</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>≠</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>∈</td>
<td>IN</td>
</tr>
<tr>
<td>∉</td>
<td>NOTIN</td>
</tr>
<tr>
<td>∨</td>
<td>∨</td>
</tr>
<tr>
<td>∧</td>
<td>∧</td>
</tr>
<tr>
<td>¬</td>
<td>¬</td>
</tr>
<tr>
<td>×</td>
<td>*</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>+</td>
<td>%</td>
</tr>
<tr>
<td>→</td>
<td>-&gt;</td>
</tr>
<tr>
<td>^</td>
<td>^</td>
</tr>
<tr>
<td>A_i</td>
<td>A!i</td>
</tr>
<tr>
<td>T^</td>
<td>T@</td>
</tr>
<tr>
<td>R</td>
<td>$R</td>
</tr>
<tr>
<td>Z</td>
<td>$Z</td>
</tr>
<tr>
<td>N</td>
<td>$N</td>
</tr>
<tr>
<td>B</td>
<td>$B</td>
</tr>
<tr>
<td>I</td>
<td>$I</td>
</tr>
</tbody>
</table>
APPENDIX D
THE FUNCTIONAL LANGUAGE—Φ
(RIGHT-RECURSIVE GRAMMAR)

Note: $(...)^*$ means zero or more occurrences
$(...)^+$ means one or more occurrences
$(...)^n$ means from zero to $n$ occurrences
$(x \mid y)$ means either $x$ or $y$, but not both

```
BLOCK        ::= BEGIN BLOCKBODY END
BLOCKBODY    ::= LET DEFS; BLOCKBODY QUALEXP
DEFS         ::= DEF (AND DEFS)$^*$
DEF          ::= (ID)$^1$ FORMALS $\equiv$ QUALEXP
               ID : TYPEEXP
               TYPE ID (FORMALS)$^1$ $\equiv$ TYPEEXP
QUALEXP      ::= EXPRESSION (WHERE AUXDEFS)$^*$
AUXDEFS      ::= AUXDEF (AND AUXDEF)$^*$
AUXDEF       ::= (ID)$^1$ FORMALS $\equiv$ EXPRESSION
FORMALS     ::= (FORMALS (FORMALS)$^*$)
               ID
EXPRESSION   ::= CONJUNCTION ( $\lor$ CONJUNCTION)$^*$
CONJUNCTION  ::= NEGATION ( $\land$ NEGATION)$^*$
NEGATION     ::= ($\neg$)$^1$ RELATION
RELATION     ::= SIMPLEXEP (RELATOR SIMPLEXP)$^1$
```
RELATOR ::= =
  ≠
  LESS
  GREATER
  ≤
  ≥
  ∈
  ∈
SIMPLEXP ::= TERM (ADDOP TERM)*
ADDOP ::= +
  -
  ≤
  ≥
  <=
  ≥
TERM ::= FACTOR (MULOP FACTOR)*
MULOP ::= *
  /
FACTOR ::= + PRIMARY
  - PRIMARY
  PRIMARY
PRIMARY ::= APPLICATION (! APPLICATION)*
APPLICATION ::= (ACTUAL)*
ACTUAL ::= ID
  DENOTATION
  CONDITIONAL
  COMPOUND
  ARGBINDING
  BLOCK
  FILE "CHAR"
  COMMENT
DENOATION ::= "CHAR"
  "DIGIT"
  "DIGIT" "DIGIT"
  "DIGIT"... "DIGIT"
  FORMALS... ACTUAL
ID ::= ALPHENUM
  ID ::= ALPHENUM... ALPHENUM
  ID ::= ALPHENUM
  ALPHENUM ::= A... Z
  ALPHENUM ::= a... z
  ALPHENUM ::= 0... 9
  ALPHENUM ::= "_"
  ALPHENUM ::= "-
  ALPHENUM ::= "_"
CONDITIONAL ::= IF ARM { ELSEIF ARM } { ELSE_EXPRESSION } ENDIF
ARM ::= { EXPRESSION THEN_EXPRESSION
  ELSE_EXPRESSION }
COMPOUND := ( (ELEMENTS)', )
            ( (ELEMENTS)', )
            < (ELEMENTS)', >

ELEMENTS := QUALEXP(QUALEXP)'

ARGBINDING := [ op ]
             [ OP QUALEXP ]
             [ QUALEXP OP ]

OP :=
    RELATOR
    ADDOP
    MULOP

TYPEREXP := TYPEDOM ( → TYPEDOM)'

TYPEDOM := TYPETERM ( + TYPETERM)'

TYPETERM := TYPEFAC ( * TYPEFAC)'

TYPEFAC := TYPEPRIMAR Y*
           TYPEPRIMAR Y
           ID <<TYPEREXP , TYPEDOM)' >>

TYPEPRIMAR Y := (TYPEREXP)
                ID
                PRIMTYPE

PRIMTYPE := R
           Z
           H
           B
           I
           TYPE

FOR INTERACTIVE IMPLEMENTATION OF Φ

SESSION := (COMMAND)'

COMMAND := (DEF | QUALEXP) ;
APPENDIX E
ROCK COMPILER HEADER FILES

******************************************************************************
* THIS FILE CONTAINS HEADER FILES REQUIRED BY THE ROCK COMPILER           *
******************************************************************************

******************************************************************************
* PUBLIC DOMAIN SOFTWARE                                                    *
******************************************************************************

* Name                  : scanner definitions                           *
* File                  : scanner.h                                     *
* Authors               : Maj E.J. COLE / Capt J.E. CONNELL            *
* Started               : 10/10/86                                     *
* Archived              : 12/11/86                                     *
* Modified              : 01/10/87 - Update keywords  JC                *
******************************************************************************

This file contains definitions used by the scanner, parser, and error recovery routines.

This file contains definitions used by the scanner, parser, and error recovery routines.
Modified : 01/10/87 Corrections to comply with latest definitions of the language and update keywords. JC

#define EOF
#define FALSE 1
#define TRUE 0
#define RTOKEN 1
#define MAX Keywords 17
#define NAMESIZE 19
#define VAXLINE 80
#define TABLESIZE 128

/* General Token Types */
/* Listing of symbols can be found at end of list */

#define IDLN 3
#define SEC 4
#define SEC 5
#define AT SEQUENCE 6
#define SEC 7
#define END SEQUENCE 8
#define C 9
#define ADE 10
#define SEC 11
#define T 12
#define W 13
#define ID 14
#define SEC 15
#define SEC 16

57
#define COMMA 7
#define LPAREN 18
#define RPAREN 19
#define EQUIV 20
#define ORLOG 21
#define ANDLOG 22
#define NEQLOG 23
#define COLON 24
#define CAT 25
#define LTBRAKET 26
#define RTBRAKET 27
#define LTSQIG 28
#define RTSQIG 29
#define EMPT_LIT 30
#define RTARROW 31
#define LITERAL 32
#define IDENTIFIER 33
#define CONSTANT 34
#define REAL 35
#define INTEGER 36
#define NATURAL 37
#define BOOLEAN 38
#define TRIVIAL 39
#define CHAR 40
#define STRING 41
#define INTEGER 42
#define POS 43
#define NEG 44
#define /* KEYWORD */

/* eof, error, unknown token, <=, =>, <>, <=, =>, =, +, - , *, /, :, ;, [ ], ( ), ) , ( , ), \, /, \, :, ^, [ , ] , ( , ) , ' , ->, literal, identifier, constant, SR, SZ, SN, SB, $I, character, string, $, unary plus, unary minus, keyword */

/* Keywords */

/* define AND */
/* define BEGIN */
/* define ELSE */
/* define ELIF */
/* define END */
/* define ENDF */
/* define FILE */
/* define GREATER */
/* define IF */
/* define IN */
/* define LESS */
/* define LET */
/* define NOTIF */
/* define READ */
/* define THEN */
/* define TYPE */
/* define WHERE */
/* define WRITE */
#define CALLCC(y,x) ((x) callcc(y, sizef(x)))

struct NStruct {
    char name: NAMESIZE;
    struct NStruct *line;
};

typedef struct NStruct NameRec;

extern char *callcc();
extern char *malloc();
PUBLIC DOMAIN SOFTWARE

* Name : parser definitions
* File : parser.h
* Authors : Maj E.J. COLE / Capt J.E. CONNELL
* Started : 10/20/86
* Archived : 12/11/86
* Modified : 01/12/87 - update NodeStruct definition JC

This file contains definitions used by the parser

Modified : 01/10/87 - update NodeStruct to hold the type of the node

#define LETTER
#define DEFANG
#define KINDER
#define FUNID
#define FUNDEF
#define CADER
#define TDEFID
#define TDEFON
#define CADAUXDEF
#define FDATADEF
#define AHEAD
#define ACTUALIST
#define SEQUENCE
#define FORMAL
#define ELIST
#define EMPTYCOMPONENT
#define EMPTYSEQUENCE
#define ANIDENT
#define ANIDENTP
#define ANIDENTT
#define TYPDEF
#define TYPDEFV
#define TYPDEFLIST
#define TYPE
#define DAS
#define DAS9
#define Node
#define NODE
typedef struct NodeStruct NodeRec, *NodeRec;  

int n errors;  
int acquired;  
char *callcol();  
char *alloc();  
ErrorHandler();  
WriteErrors();

/*********************** External Utility Functions *******************/

extern NodeRec *CreateNode();
extern char *NodeName();
extern MakeNewRoot();
extern (Format());
extern (Ball());
extern (End());
extern (Long ByPass());

#include <scanner.h>
#include <errors.h>

*end;
PUBLIC DOMAIN SOFTWARE

Name: error file definitions
File: errors.h
Authors: Maj E.J. COLE / Capt J.E. CONNELL
Started: 01/20/87
Archived: 04/07/87
Modified:

This file contains definitions used by the error recovery routines.

 ifndef MAXERRORS
#define MAXERRORS 10

/**************************** PARSER ERRORS ****************************/

#define ERR0 0  /**< ' ' or '"' w/o '"' */
#define ERR1 1  /**< ' ' w/o '/' -- bad logical OR */
#define ERR2 2  /**< 'S' w/o proper following char */
#define ERR3 3  /**< invalid numeric constant */
#define ERR4 4  /**< literal w/o ending */
#define ERR5 5  /**< unidentified char in input file */
#define ERR6 6  /**< out of memory */
#define ERR7 7  /**< error in statement following '
' */
#define ERR8 8  /**< error in type definition following '
' */
#define ERR9 9  /**< error in type definition */
#define ERR_a 10 /**< following 'xx' */
#define ERR_b 11 /**< unable to complete eval of */
#define ERR_c 12 /**< missing or misplaced ; after */
#define ERR_d 13 /**< invalid QualExp */
#define ERR_e 14 /**< invalid TypeExp */
#define ERR_f 15 /**< bad or missing forms */
#define ERR_g 16 /**< missing or misplaced */
#define ERR_h 17 /**< missing ID after 'TYPE' */
#define ERR_i 18 /**< bad definition after AND */
#define ERR_j 19 /**< missing or misplaced '1' */
#define ERR_k 20 /**< error in processing */
#define ERR_l 21 /**< successive Actuels */
#define ERR_m 22 /**< missing literal after keyword */
#define ERR_n 23 /**< FILE */
#define ERR_o 24 /**< missing or invalid exp after */
#define ERR_p 25 /**< keyword */
#define ERR_q 26 /**< IF statement w/o ELSE */
#define ERR_r 27 /**< error in forms preceding -- */
#define ERR_s 28 /**< following comp op */
#define ERR_t 29 /**< error in ArgBinding - check */
#define ERR_u 30 /**< off in DZCNLsimple-erred */
#define ERR_v 31 /**< feature */

62
```c
#define ERR_s 28 /* */
#define ERR_t 29 /* */
#define ERR_u 30 /* */
#define ERR_v 31 /* */
#define ERR_w 32 /* */
#define ERR_x 33 /* */
#define ERR_y 34 /* */
#define ERR_z 35 /* */

/* NOTE: s through z reserved for future use */

/**************************** SEMANTIC ERRORS ****************************/

#define ERR_aa 35 /* Numeric value expected */
#define ERR_bb 35 /* Natural expected */
#define ERR_cc 35 /* Integer or natural expected */
#define ERR_dd 35 /* Error in Tuple Definition */
#define ERR_ea 35 /* Undefined var in "and" scope */
#define ERR_ff 35 /* Function w/o function def */
#define ERR_gg 35 /* Formais mismatch */
#define ERR_hh 35 /* Undefined function */
#define ERR_ii 35 /* Real Number expected */
#define ERR_jj 35 /* Invalid Constant */
#define ERR_kk 35 /* Boolean value Expected */
#define ERR_ll 35 /* Boolean Operator Expected */
#define ERR_mm 35 /* Out of run-time memory space */

#endif
```
/* Name : Semantic Definitions Header File */
/* File : Semcheck.h */
/* Authors : Maj E.J. COLE / Capt J.E. CONNELL */
/* Started : 01/01/87 */
/* Archived : 04/10/87 */
/* Modified : 04/13/87 "FILENAME" eliminated EC */

This file contains the header file and definitions for the semantic
checker and code generator of the PHI compiler

* Modified : 04/13/87 "FILENAME" eliminated; output path now
* depends on user's input EC

*****************************************************************************/

/*******************************************~*************** *****/

externals
***************** ********
#include <scanner.h>
#include <parser.h>
#include <errors.h>
#include <stdio.h>

*******************************************************************************/

globals
****************************
#define NOTFOUND 0    /* Definition for findvar */
#define UNTYPED 0     /* Type Definitions and sizes */
#define BOOLEAN 1
#define BOL_BYTES 2
#define REAL 2
#define REALBYTES 4
#define INTEGER 3
#define INT_BYTES 2
#define NATURAL 4
#define NATBYTES 2
#define ERROR 0
#define MAXADDR 64000  /* Max # of bytes in var space */
#define MAXTYPES 300   /* Max # of types in one scope */
#define CODE_SIZE 20000 /* Max size of code buffer */
#define START_ADDR 0   /* Starting address for varspace */
#define TYPE_INIT 5    /* Pointer to the last initial */
#define CNTRL Z 26     /* Control Z ascii */
#define ENDS-RING 0    /* String terminator */
#define NUM_BASE 48    /* Lowest ascii number */
#define STACKSIZE 10000 /* Increase in stack size */
#define SIZEBUFFER 30000 /* Size of output buffer */
#define ADD 1
#define SUB 2
#define DIVIDE 3
#define MUL 4
#define SEM_ERR 0    /* Flag to indicate semantic */
#define NULL '0'

endif
Type Definitions

typedef int optype,
    FLAG,
    N'TYPE;

typedef char stq [20];

typedef struct and_struct *and_ptr;

Typetable Definitions

typedef struct typenode {
    char name [10];
    int bytes;
    struct typenode *typeptr;
} tnode;

Formallist Definitions

typedef struct formnode {
    int name, type;
    struct formnode *link;
} fnode;

Variable Definitions

typedef struct varnode {
    int type,
    form,
    def;
    nodal nptr;
    fnode *fptr;
    struct varnode *link;
    *varptr;
} vnode;

Deftable Definitions

typedef struct defnode {
    int type;
    nodal nptr;
    fnode *fptr;
    struct defnode *link;
    *defptr;
} dnode;

And Definitions

struct and_struct {
    (nodal ptr;
    int buffptr;
    struct and_struct *link;
};
/**
 * PUBLIC DOMAIN SOFTWARE
 *
 * Name : User Header
 * File : user.h
 * Authors : Maj E.J. COLE / Capt J.E. CONNELL
 * Started : 04/01/87
 * Archived : 04/10/87
 * Modified :
 *
 * This file is the header file for the user interface module
 * (user.c)
 */

**************************************************************************
/*
** Globals
**************************************************************************/

#define BUFFLENGTH 30 /* Max size of input file name.
#define NOTFOUND 0 /* directory.
#define BSIZE 1000 /* Input buffer size.
#define BLOCKSIZE 50 /* Input block size.
#define BACKSPACE 8 /* ASCII Equivalents.
#define EOLN 13
#define ESCAPE 27
#define GETPROGRAM "Program to Compile -> " /* Messages to observer.
#define HEADER1 "ROCK COMPILER"
#define HEADER2 "Press Escape Key to Exit Compiler"
#define FILE1_ERROR "File not Found"
#define FILE2_ERROR "Press ESCAPE to exit, any other key to continue"
#define WAIT "Compiling: Please Wait"
#define FAUSE "PRESS ANY KEY TO CONTINUE"

#define ERRORFILE "errors.phi" /* Textfile of errors.

66
APPENDIX F
ROCK COMPILER — MAIN MODULE

/* PUBLIC DOMAIN SOFTWARE */

* Name : Main Rock Module
* File : Rock_main.c
* Authors : Maj E.J. COLE / Capt J.E. CONNELL
* Started : 01/06/87
* Archived : 04/10/87
* Modified : 04/13/87 Output files put to vdisk EC

This file contains the following modules for the PHI compiler:

R_Initial  Semcheck  Main

Algorithm :
This contains the main procedure for the phi compiler, in addi-

tion to the initialization procedure & the main semantic checking
procedure. The main module inits the program, sets up the screen
by calling "user ()", & decides whether an error routine needs

to be called. It also closes out the input file.

The "semcheck procedure is designed to be called by any function
with a ptr to a parse tree node as an argument. It will then
determine which sub-module is necessary to check the node.

"R_Initial" presently has the function of initializing the type

*table.

* Modified : 04/13/87 Output files written to vdisk, "d:" EC

*******************************************************************************

Externals
*******************************************************************************

#include (semcheck.h)

extern void g_startup (),
  g_start ()
extern "C" :
  int c91 ()
  char *c92 ()
  long c93 ()
  long c94 ()

extern "C" :
  int c91 ()
  char *c92 ()
  long c93 ()
  long c94 ()

*******************************************************************************

Globals
*******************************************************************************

unsigned _stack = STACKSIZE.

67
/********************** R_Initial **********************/

void
r_initial ()
{extern nodo types [];

strcpy (types [UNTYPED].name, "untyped");
types [UNTYPED].bytes = NULL;
strcpy (types [BOOLEAN].name, "boolean");
types [BOOLEAN].bytes = Bool_BYTES;
strcpy (types [REAL].name, "real");
types [REAL].bytes = REAL_BYTES;
strcpy (types [INTEGER].name, "integer");
types [INTEGER].bytes = INT_BYTES;
strcpy (types [NATURAL].name, "natural");
types [NATURAL].bytes = NAT.Bytes;

/********************** SemChecker **********************/

PHITYPE
semcheck (ptr)
{nodal ptr;
{extern PHITYPE tkindf (), trarrow (),
tfunid (), tid (), tconstant (), tactuallist (), tactuals, p;
PHITYPE type;

switch (ptr->name) {
  case (ADD_):
  case (SUB_):
  case (MULT_):
  case (RDIV_):
  case (IDIV_):
  case (COLON_):
  case (CAT_):
    type = arithop (ptr);
    break;
  case (POS_):
  case (POD_):
  case (ORLOG_):
  case (ANDLOG_):
  case (NEGLOG_):
  case (KINDEF):
  case (RTARROW_):
  case (LETDEF):
  case (KW_ + WHERE_):
  case (AUXAND):
  case (DATAAUXDEF):
  case (FUNAUXDEF):
  case (FUNID):
  case (ACTUALLIST):
    break;
  case (COMMA_):
    break;
}
APPENDIX G
ROCK COMPILER — SCANNER

PUBLIC MAIN SOFTWARE

**Name:** Scanner

**Use:**

**Keywords:**

- **KeyWord:**

**Methods:**

- **Keyword:**

**Variables:**

- **Type:**

**Examples:**

- **Example:**

**Practical Application:**

- **Application:**

**Conclusion:**

- **Conclusion:**
return LAST_SEQUENCE;

case 'A':
  return getstructure("A");
  return VALUE;
else
  lookahead = 0;
  return NULL;

case 'B':
  return getstructure("B");
  return VALUE;
else
  lookahead = 0;
  return NULL;

// Similar cases...
CHEXKS TO SEE IF THE INPUT TOKEN IS A KEYWORD IN THE LANGUAGE.
* IF IT IS, THE FUNCTION RETURNS THE NUMERIC VALUE OF THE KEYWORD.
* IF IT ISN'T, THE FUNCTION RETURNS -1. PERFORMS BINARY SEARCH OF
  KEYWORD ARRAY.
* MUST KEEP THIS ARRAY IN ALPHABETICAL ORDER.

MAX KEYWORDS = 33
"AND", "RETURN", "TO", "FOR", "DO", "WHILE", "IF", "GOTO", "FOR", "REPEAT", "THEN", "TYPE", "WHERE", "PRINT";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"and", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
"case", "if", "then", "else", "while", "do", "for", "return", "and", "goto";
APPENDIX H
ROCK COMPILER — PARSER

PUBLIC DOMAIN SOFTWARE

* Name : parser pt I
* File : parser1.c
* Authors : Maj E.J. COLE / Capt J.E. CONNELL
* Started : 10/20/86
* Archived : 12/11/86
* Modified : 04/23/87 No longer set up to work with file of tokens.

* This file contains the following modules for the PHI parser:

BlockBody() LetDefs() Defs() DefAnd() QualExp()
AuxExp() AuxDefs() AuxAnd() Formals() Expression()

* Algorithm : The main module calls BlockBody() to start the parse
off. BlockBody in turn calls LetDefs() first and then
QualExp() looking for a valid program. The remaining
modules in Pt's 1-3 are called by these when trying to
validate a program. The results from the parse are now
kept in an abstract syntax tree for type checking and
code generation. Various utility functions are used
to build the tree and simplify parsing the grammar.

* Modified : 12/26/86 Flattened tree output changed to abstract
  syntax tree form. JC
  01/10/87 Corrections to comply with latest definitions
  of the language. JC
  01/27/87 Error Recovery added and files combined. JC
  03/20/87 Token buffer implemented for parser. JC
  03/29/87 Changed manner errors are handled — required
  for integration with back-end.
  04/23/87 No longer set up to work with file of tokens.
  GetToken is called directly thru FillBuff(). JC

#include <stdlib.h>
#include <parser.h>

int Global = FALSE, arabin = FALSE;

// APPENDIX H
// ROCK COMPILER — PARSER

PUBLIC DOMAIN SOFTWARE

* Name : parser pt I
* File : parser1.c
* Authors : Maj E.J. COLE / Capt J.E. CONNELL
* Started : 10/20/86
* Archived : 12/11/86
* Modified : 04/23/87 No longer set up to work with file of tokens.

* This file contains the following modules for the PHI parser:

BlockBody() LetDefs() Defs() DefAnd() QualExp()
AuxExp() AuxDefs() AuxAnd() Formals() Expression()

* Algorithm : The main module calls BlockBody() to start the parse
off. BlockBody in turn calls LetDefs() first and then
QualExp() looking for a valid program. The remaining
modules in Pt's 1-3 are called by these when trying to
validate a program. The results from the parse are now
kept in an abstract syntax tree for type checking and
code generation. Various utility functions are used
to build the tree and simplify parsing the grammar.

* Modified : 12/26/86 Flattened tree output changed to abstract
  syntax tree form. JC
  01/10/87 Corrections to comply with latest definitions
  of the language. JC
  01/27/87 Error Recovery added and files combined. JC
  03/20/87 Token buffer implemented for parser. JC
  03/29/87 Changed manner errors are handled — required
  for integration with back-end.
  04/23/87 No longer set up to work with file of tokens.
  GetToken is called directly thru FillBuff(). JC

#include <stdlib.h>
#include <parser.h>

int Global = FALSE, arabin = FALSE;

// APPENDIX H
// ROCK COMPILER — PARSER

PUBLIC DOMAIN SOFTWARE

* Name : parser pt I
* File : parser1.c
* Authors : Maj E.J. COLE / Capt J.E. CONNELL
* Started : 10/20/86
* Archived : 12/11/86
* Modified : 04/23/87 No longer set up to work with file of tokens.

* This file contains the following modules for the PHI parser:

BlockBody() LetDefs() Defs() DefAnd() QualExp()
AuxExp() AuxDefs() AuxAnd() Formals() Expression()

* Algorithm : The main module calls BlockBody() to start the parse
off. BlockBody in turn calls LetDefs() first and then
QualExp() looking for a valid program. The remaining
modules in Pt's 1-3 are called by these when trying to
validate a program. The results from the parse are now
kept in an abstract syntax tree for type checking and
code generation. Various utility functions are used
to build the tree and simplify parsing the grammar.

* Modified : 12/26/86 Flattened tree output changed to abstract
  syntax tree form. JC
  01/10/87 Corrections to comply with latest definitions
  of the language. JC
  01/27/87 Error Recovery added and files combined. JC
  03/20/87 Token buffer implemented for parser. JC
  03/29/87 Changed manner errors are handled — required
  for integration with back-end.
  04/23/87 No longer set up to work with file of tokens.
  GetToken is called directly thru FillBuff(). JC

#include <stdlib.h>
#include <parser.h>

int Global = FALSE, arabin = FALSE;

// APPENDIX H
// ROCK COMPILER — PARSER

PUBLIC DOMAIN SOFTWARE

* Name : parser pt I
* File : parser1.c
* Authors : Maj E.J. COLE / Capt J.E. CONNELL
* Started : 10/20/86
* Archived : 12/11/86
* Modified : 04/23/87 No longer set up to work with file of tokens.

* This file contains the following modules for the PHI parser:

BlockBody() LetDefs() Defs() DefAnd() QualExp()
AuxExp() AuxDefs() AuxAnd() Formals() Expression()

* Algorithm : The main module calls BlockBody() to start the parse
off. BlockBody in turn calls LetDefs() first and then
QualExp() looking for a valid program. The remaining
modules in Pt's 1-3 are called by these when trying to
validate a program. The results from the parse are now
kept in an abstract syntax tree for type checking and
code generation. Various utility functions are used
to build the tree and simplify parsing the grammar.

* Modified : 12/26/86 Flattened tree output changed to abstract
  syntax tree form. JC
  01/10/87 Corrections to comply with latest definitions
  of the language. JC
  01/27/87 Error Recovery added and files combined. JC
  03/20/87 Token buffer implemented for parser. JC
  03/29/87 Changed manner errors are handled — required
  for integration with back-end.
  04/23/87 No longer set up to work with file of tokens.
  GetToken is called directly thru FillBuff(). JC

#include <stdlib.h>
#include <parser.h>

int Global = FALSE, arabin = FALSE;

// APPENDIX H
// ROCK COMPILER — PARSER

PUBLIC DOMAIN SOFTWARE

* Name : parser pt I
* File : parser1.c
* Authors : Maj E.J. COLE / Capt J.E. CONNELL
* Started : 10/20/86
* Archived : 12/11/86
* Modified : 04/23/87 No longer set up to work with file of tokens.

* This file contains the following modules for the PHI parser:

BlockBody() LetDefs() Defs() DefAnd() QualExp()
AuxExp() AuxDefs() AuxAnd() Formals() Expression()

* Algorithm : The main module calls BlockBody() to start the parse
off. BlockBody in turn calls LetDefs() first and then
QualExp() looking for a valid program. The remaining
modules in Pt's 1-3 are called by these when trying to
validate a program. The results from the parse are now
kept in an abstract syntax tree for type checking and
code generation. Various utility functions are used
to build the tree and simplify parsing the grammar.

* Modified : 12/26/86 Flattened tree output changed to abstract
  syntax tree form. JC
  01/10/87 Corrections to comply with latest definitions
  of the language. JC
  01/27/87 Error Recovery added and files combined. JC
  03/20/87 Token buffer implemented for parser. JC
  03/29/87 Changed manner errors are handled — required
  for integration with back-end.
  04/23/87 No longer set up to work with file of tokens.
  GetToken is called directly thru FillBuff(). JC

#include <stdlib.h>
#include <parser.h>

int Global = FALSE, arabin = FALSE;
```c
/* must use "long" because buffer holds addresses */
/* use BUFSIZE = 1 in case user's */
/* place address of "fre" at end */
/* of buffer */

and tokenbuff BUFSIZE-1, *ptr = tokenbuff BUFSIZE;
FILE *postfile, *errorfile; /* working files */

FLICTPASE();

int root = NULL;
extern void parse(int, mov current);
int errors = 0;

errorfile = fopen("errors.err", "w");
printf(errorfile, "%s
", "ROCKY ERRORS
");
fclose(errorfile);

#define DEBUG
postfile = fopen("parser.out", "w");

/* look for a valid program */
/* set cursor at start */
/* treat extraction, fail case */
/* end of file */

printf("FAIL
");

exit(EXIT_FAILURE);

/* do the extraction */
/* post file */

	/* this is real */
/* else */
	/* error */
		/* post */
		/* error */
		/* exit */

	/* post */
```

- Does a post order walk of the tree with (root) as its head.
- Just prints out the node name to the screen now.
```c
static int l = 0;

if (root != NULL)
  PostOrder(root->lptr);
PostOrder(root->rptr);

switch (root->name)
  case IDENTIFIER:
  case CONSTANT:
  case LITERAL:
    printpositive("%d ", root->name);
    for (i = 0; root->index[i]; i++)
      break;
    default:
      printpositive("%s ", root->name);
    
    if (l == 0) printpositive("|
```

```c
      *c:~ icase
```

```c
static root;
Named *root;

* BLOCK BODY = (DEF EXP) = ""* "" 
```

```c
* exp:
  * flag = setDef root = ID Exp
  * return EXP;
```

```c
case flag = 3996
exp = J.IA Exp

* return flag;

```
<DEFAND> ::= <DEF> <DEFS>  
* and <DEFS> "need not be present."
* 
Note: This function assumes root is not NULL upon entry
*

- (class KW - AND,)
  <expr>(root, DEFAND, LEFT);   // found "and" so fix tree
  if (!expr & !expr->expr) { TRUE)
    return(expr->expr, ERR, 
    or1(expr->expr, SEMI));    
  // note it, try to fix
  end ByPass AND
  end DefAnd

* <QUALEXP> ::= <EXPRESSION> where <AUXEXP>  
* Where "where <AUXEXP>" need not be present.
* 

- i.e., entered "": scan("%c");

* errors already reported;  
  error("MK-AND",
  if (MK WHERE (WHERE))
  return(expr->root, MK WHERE, LEFT);
  expr->root->ptr = NULL;
  * end ByPass WHERE

* Qualex expressed: min, flag;

* default - just return i.e.;
  end Qualex

* root is a ptr to tree subtree
  currently working with
  
<EXPRESSION> ::= <AUXDEF>(where <AUXEXP>)
int flag;

if(flag = AuxDefs(root))!= TRUE) /* need at least one AUXDEF */
  ErrorHandler(line_no,ERR_1,
  (long)KW_WHERE_);

if(Bypass(KW_ WHERE_)) /* looking for multiple WHERE's */
  MakeNewRoot(root,(KW_ WHERE_),RIGHT);
  AuxExp(&(*root)->rptr));

return(flag); /* default - return result of */
/* first AuxDefs */
/* end AuxExp */

**************************************************************************

AuxDefs(root) /* root is a ptr to tree/subtree */
  AuxRec **root;

  <AUXDEFS> ::= (<DATAAUXDEF> | <FUNAUXDEF>) <AUXAND> */
  Where "<AUXAND> " need not be present.

select **temp; /* address of data struct holding */
  flag; /* identifier name */
  ptr; /* set up its side of subtree */

  if(Bypass(IDENTIFIER_)) /* looking for ID == */
    temp = CreateNode(IDENTIFIER_); /* found '==' It's a DATAAUXDEF */
    temp->index = ptr;
    /* Expressions(*(*root)->rptr)) := TRUE) */
    ErrorHandler(line_no,ERR_1,
    (long)KW_ WHERE_);
    /* end ByPass EQUV. */
    temp->expr = temp;
    if(EQUV_)
      root = CreateNode(DataAUXDEF);
      root->lptr = temp;
      /* Expression(*(*root)->rptr)) := TRUE) */
      ErrorHandler(line_no,ERR_1,
      (long)KW_ WHERE_);
      /* end ByPass EQUV. */
      /* not '==' so must be ID FORMALS */
      temp->Expr = temp;
      if(FUNAUXDEF)
        temp->ptr = CreateNode(FUNAUXDEF);
        /* Expressions(*(*root)->rptr)) := TRUE) */
        ErrorHandler(line_no,ERR_1,
        (long)KW_ WHERE_);
        /* end ByPass EQUV. */
        /* will look for ID FORMALS */
        temp->Exp = temp;
        if(FUNAUXDEF)
          temp->ptr->lptr = temp;
          /* Expression(*(*root)->rptr)) := TRUE) */
          ErrorHandler(line_no,ERR_1,
          (long)KW_ WHERE_);
          /* end ByPass EQUV. */
          /* note, try to fix */
          flag = 1;
          /* looking for '->',already */
          if(FUNAUXDEF)
            temp->Exp = temp;
            /* need QualExp on it */
            note the errors, try & fix
            
            /* '->' */

            /* '->' */
/* * 
** 
** This function assumes root is not NULL upon entry 
** 
*/

void 
AuxAnd(root)  
NodeRec *root; 
/* root is a ptr to tree/subtree */
#ifdef 
/* 
*/
/* 
** Where "and <AUXDEFS>" need not be present. 
** 
*/
#endif 

int 
Formals(root)  
NodeRec *root; 
/* root is a ptr to tree/subtree */
#ifdef 
/* 
*/
/* 
*/
#endif

if ((ptr = ByPass(IDENTIFIER))) 
if (ByPass(KW_AND_))
MakeNewRoot(root, AUXAND, LEFT); 
if ((AuxDefs(&(*root)->rptr) != TRUE))
ErrorHandler(line_no, ERR_h,
            (long)KW_AND_); 
/* found "and" so fix tree 
* note it, try & fix 
*/
/* ByPass AND 
*/
/* AuxAnd 
*/
/* root is a ptr to tree/subtree */
/* currently working with */
/* temp ptrs to nodes in tree 
*/
/* workingptr marches down the 
*/
/* rt side of the subtree */
/* checking for just an ID */
```c
/* Expression(root) */
NodeRec **root;

/ * <EXPRESSION> ::= <CONJUNCTION> ( \ / <EXPRESSION>) * /

int flag = 0;

if(((flag = Conjunction(root)) == TRUE)) /* look for Conjunction */
if(ByPass(ORLOG_)) /* will recursively check for */
   MakeNewRoot(root,ORLOG_,LEFT); /* found, so fix root for return */
if(Expression(((root)->rptr)) == TRUE)) /* \ / w/o following Exp. */
   ErrorHandler(line_no,ERR8, (long)ORLOG_); /* Just note it, no fix */
   return(ERROR_); /* end recursive search */

return(flag); /* end Expression() */
/ ************************************************************/
PUBLIC DOMAIN SOFTWARE

Name: parser pt 2
File: parser2.c
Authors: Maj. E. J. COLE / Capt. J. E. CONNELL
Started: 10/20/86
Archived: 12/11/86
Modified: 01/27/87 - Error Recovery added. JC

This file contains the following modules for the PHI parser:

- Conjunction()
- Negation()
- Relation()
- Relator()
- SimpExp()
- AddOp()
- Mullop()
- Term()
- Factor()
- Primary()
- Application()
- Actual()

Algorithm: See parser part 1

Modified: 12/26/86 Flattened tree output changed to abstract syntax tree form. JC
: 01/10/87 Corrections to comply with latest definitions of the language. JC
: 01/27/87 Error Recovery added and files combined. JC

```c
#include <stdio.h>
#include <parser.h>

extern int line_no;         /* global var, holds current line */
extern int rtnbrkt;         /* global flag - aids in making */
                         /* PHI deterministic */
/

int Conjunction(root)    /* root is a ptr to tree/subtree */
    NodeRec **root;    /* currently working with */

/*
<CONJUNCTION> ::= <NEGATION> ( /\ <CONJUNCTION>) */

int flag;

if ((flag * Negation(root)) == TRUE)    /* look for Negation part */
    if (ByPass(ANDLOG))    /* will recursively check for /\ */
        MakeNewRoot(root, ANDLOG_LEFT);    /* found, fix root for return */
    if (Conjunction(((root)->rptr)) == TRUE)    /* /\ w/o following Neg. */
        ErrorHandler(line_no, ERR8,        /* just note it, no fix */
                        (long)ANDLOG);    /* */
        return(ERROR_);        /* end recursive search */
    return(flag);        /* end Conjunction() */

/* */

int Negation(root)    /* root is a ptr to tree/subtree */
    NodeRec **root;    /* currently working with */

85
```c

/*  NEGATION  */

// Parse NEGATION

/*  RELATION  */

// Parse RELATION

/*  SIMPLEEXP  */

// Parse SIMPLEEXP

Where <RELATOR><SIMPLEXP> need not be present.

/*<RELATOR> ::= = | <> | < | > | <= | >= | in | not in 
Note: returns the Relator value vise TRUE if found */

int Relator();
```
...
/* flag = Actual(root) */
NodeRec **root;

if(flag = Actual(root) == TRUE)
   /* look for an actual */
   if(flag = Application(&tnode) == TRUE)
      /* look for an actual list */
      MakeNewRoot(root, ACTUALLIST, LEFT);
      (*root) -> rptr = tnode;
      if((*root) -> rptr -> name == ACTUALLIST)
         MakeNewRoot(&((*root) -> rptr),
                ACTUALLIST, LEFT);
      else if(flag == ERROR_)
         ErrorHandler(line_no, ERR_K, NULL);
         /* note it, no fix */
else return(TRUE);
return(flag);

int Actual(root)
NodeRec **root;

/* root is a ptr to tree/subtree */

/* currently working with */

/* <ACTUAL> ::= <ID> | file<LITERAL>|<CONDITIONAL>|<BLOCK>| */
/* <DENOTATION>|<COMPOUND>|<ARGBINDING> */

long ptr;
NodeRec **temp;
int flag;

if((ptr = ByPass(IDENTIFIER_)))
   /* checking for ID */
root = CreateNode(IDENTIFIER_);
(*root) ->*name = ID;
if (ByPass(LINERTARROW_))
    MakeNewRoot(root, LINERTARROW, LEFT);
    if((Actual(LINERTARROW)) = TRUE)
        return(TRUE);
    else
        if (ByPass(KW_ FILE_))
            *root = CreateNode(KW_ FILE_);
            (Actual(KW_ FILE_)) = TRUE;
            return(TRUE);
        else
            ErrorHander(line_no, ERR8, (long)LINERTARROW_); 
            return(ERROR_);
    return(TRUE);

if (ByPass(LINERTARROW_))
    return(TRUE);
else
    temp = *root;
    if( *IsFormal(temp))
        ErrorHandler(line_no, ERR8, NULL);
        (*root) ->*name = FORMAL;
        MakeNewRoot(root, LINERTARROW_ LEFT);
        if((Actual(LINERTARROW)) = TRUE)
            return(TRUE);
    else
        ErrorHander(line_no, ERR8, (long)LINERTARROW_);
        return(ERROR_);

    else if(flag == ERROR_)
        return(ERROR_);
if (!flag && !node(q)) return(flag);

if (!flag && !ArgBind.node(q)) return(flag);

return(false); /* Default, tried everything else */
/* end Actual() */

*******************************************************************************/
PUBLIC DOMAIN SOFTWARE

* Name : parser pt 3
* File : parser3.c
* Authors : Maj E.J. COLE / Capt. J.E. CONNELL
* Started : 10/20/86
* Archived : 12/11/86
* Modified : 01/27/87 - Error Recovery added. JC

This file contains the following modules for the PHI parser:

- Conditional()
- Arm()
- Block()
- Compound()
- Elements()
- Denotation()
- ArgBind()
- Op()
- TypeExp()
- TypeDom()
- TypeTerm()
- TypeFacw()
- TypePrimary()
- PrimType()

Algorithm : See parser part 1

Modified : 12/26/86 Flattened tree output changed to abstract syntax tree form. JC
*: 01/10/87 Corrections to comply with latest definitions of the language. JC
*: 01/27/87 Error Recovery added and files combined. JC

```
#include <stdio.h>
#include <parser.h>

extern int rtarket;  /* global flag - aids */
extern int line_no;   /* global var, current line */

int Conditional(root)

NodeRec **root;  /* root is a ptr to tree sub-tree */

/* <CONDITIONAL> ::= if <ARM> (elsif<ARM>)* (else<EXPRESSION>)1 endif */
NodeRec *temp = NULL, *subroot, *workingptr;  /* ptrs to temp nodes in the tree */

if(ByPass(KW_ "IF_"))
  if(Arm(*temp) != TRUE)
    ErrorHandler(line_no,ERR_m,(long)IF_);  /* note it, try to fix */
    *root = CreateNode(KW_ "IF_");
    (*root) ->lptr = temp;
    workingptr = *root;
    while(ByPass(KW_ "ELSIF_"))
      subroot = CreateNode(KW_ "ELSIF_");
      workingptr ->rptr = subroot;
      if(Arm(*temp) != TRUE)
        ErrorHandler(line_no,ERR_m, (long)ELSIF_);
        *subroot = temp;
        workingptr = workingptr ->rptr;
      if(ByPass(KW_ "ELSE_"))
```

92
<ARM> ::= <EXPRESSION>then<EXPRESSION>
/*
flag:
NodeRec **root; /* root is a ptr to tree subtree */
currently working with */
/*
<ARM> ::= <EXPRESSION>then<EXPRESSION> */

if (flag = Expression(temp)) == TRUE) /* if an error try to recover by */
ErrorHandler(line_no, ERR mơ, (long)THEN_); /* look for THEN,ELSE,ELSE1,ELSE2 */
else
ErrorHandler(line_no, ERR_f, (long)KW_THEN_); /* report it and try to press on */
return(flag); /* end Arm() */
/* ***********************************************/

/*
<BLOCK> ::= begin <BLOCKBODY> end */

if (ByPass(KW_ BEGIN_)) /* root is a ptr to tree subtree */
   root = CreateNode(KW_ BEGIN_); /* have already been reported */
   if (BlockBody((*(root) -> lptr)) == TRUE) /* look for BLOCKBODY */
"<COMPOUND> ::= '<ELEMENTS>' '</ELEMENTS>' '<ELEMENTS>'
where <ELEMENTS> may be empty"

if (troot == NIL)
  troot = CreateNode(EMPTYCOMPOUND);
else if (troot->name == COMMA)  
  troot->name = ELLIST;
return(TRUE);

if (ByPass(STSEQUENCE_))
  Elements(troot);
if (!ByPass(ENDSEQUENCE_))
  ErrorHandler(line_no, ERR_f, (long)ENDSEQUENCE_);
if (troot == NIL)  
  troot = CreateNode(EMPTYSEQUENCE);
else MakeNewRoot(troot, SEQUENCE, RIGHT);
return(TRUE);
return(FALSE);
int Elements(NodeRec **root) 
* /* root is a ptr to "tree" ... */ */ <ELEMENTS> ::= <QUALEXP> (<,<QUALEXP>) < INT flag; if(flag == QualExp(root)) == ERRBR_ 
* /* errors already set here */ */ EatEm(COMMA_); 
* /* recursively look for root */ */ while(Bypass(COMMA_)) 
* /* found a COMMA */ */ MakeNewRoot(root,COMMA_ LEFT); 
* /* fix tree so far */ */ if (Elements((root)->rptr)) == TRUE) 
* /* note it, try again */ */ ErrorHandler(line_no,ERR_p, 
* (Long)COMMA_); 
if((*root)->rptr->name == COMMA_ 
* /* hang to the leaf */ */ MakeNewRoot(((*root)->rptr), 
* COMMA_ LEFT); 
* /* end while Bypass */ */ return(flag); 
* /* end Elements */ */

******************************************************************************

int Denotation(NodeRec **root) 
* /* root is a ptr to */ */<DENOTATION> <LITERAL> | <CONSTANT> <FORMALS> | <ACTUAL> 
* /* where LITERAL is quoted(') string of zero or more characters */ */ <NOTE: <FORMALS> | -> <ACTUAL> was already checked by */ 
Long ptr;
* long */ if(ptr = Bypass(LITERAL_) 
* root = CreateNode(LITERAL_); 
* (*root) ->index = ptr; 
* return(TRUE); 
* 
if(Bypass(EMPTY_LIT_.) 
* root = CreateNode(LITERAL_); 
* (*root) ->index = NULL; 
* return(TRUE); 
* 
if(ptr = Bypass(CONSTANT) 
* root = CreateNode(CONSTANT) 
* (*root) ->index = ptr; 
* return(TRUE); 
* 
******************************************************************************
IMPLEMENTATION OF A COMPILER FOR THE FUNCTIONAL PROGRAMMING LANGUAGE PHI(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA E J COLE ET AL. JUN 87

UNCLASSIFIED
```c
/* ARGBINDING */
	/* temp ptr to node in tree */
	/* global flag needed to make */
	/* PHI deterministic */
	/* set global flag, needed to */
	/* PHI deterministic */
	
int special_case;
NodeRec *temp = NULL;
extern int argbind;

if (ByPass(RTBRAKE_))
    argbind = TRUE;
special_case = (!Ball(ADD, 1) && Ball(SUB, 1));

#define PEBUS
#define special_case "a argbind = " special_case " argbind;

endif

if (Op(root))
    if (ByPass(RTBRAKE_))
        argbind = FALSE;
        MakeNewRoot(root, ARGBINDOP, LEFT);
        return(TRUE);
    MakeNewRoot(root, ARGBINDOP, LEFT);
    if (!Ball(ADD, 1) && Ball(SUB, 1))
        special_case = FALSE;
    if ((QualExp((*root)->rptr) == TRUE)
        if (ByPass(RTBRAKE_) == TRUE)
            argbind = FALSE; return(TRUE); /* reset global flag */
    else
        if (specialcase && Op(temp))
            if (ByPass(RTBRAKE_))
                ((*root) = lptr) = rptr = NULL;
            (*root) = rptr = temp;
            if ((lptr) = name = ADD
                || (lptr) = name = POS
                || (lptr) = name = NEG
                || (*root) = name = ARGBINDOP
                )
                argbind = FALSE;
                return(TRUE);
            else
                argbind = FALSE;
                ErrorHandler(line_no, ERR_q, NULL);
                return(ERROR_);
        else
            if ((QualExp(root)) == FALSE)
                MakeNewRoot(root, ARGBINDOP, LEFT);
                argbind = FALSE;
                if (Op1(*root) = rptr)
                    if (ByPass(RTBRAKE_))
                        return(TRUE);
                ErrorHandler(line_no, ERR_q, NULL);
                return(ERROR_);
            return(FALSE);

    /* end else specialcase && Op() */
    /* & RTBRAKE_ */
    /* end 2 cases where QualExp TRUE */
    /* reset global flag */
    /* report it, no fix */
endif

/* end if QualExp comes first */
/* end if ByPass LTBRAKE_ */
/* default, none of the above */
/* end ArgBinding() */
```

/* root is a ptr to tree/subtree */
NodeRec **root; /* currently working with */

/* <OP> ::= , | ! | <RELATOR> | <ADDOP> | <MULOP> */

int flag;
if(flag == ByPass(COMMA_))
  *root = CreateNode(COMMA_);
else if(flag == ByPass(SUBSCRIPT_))
  *root = CreateNode(SUBSCRIPT_);
else if(flag == Relator())
  *root = CreateNode(flag);
else if(flag == AddOp())
  *root = CreateNode(flag);
else if(flag == MulOp())
  *root = CreateNode(flag);

return(flag);

/***************************************************************************/

int TypeExp(root)
  NodeRec **root; /* root is a ptr to tree/subtree */
  /* currently working with */

/* <TYPEEXP> ::= <TYPEDOM> ( -<TYPEEXP> ) */
NodeRec *newroot; /* temp ptr to nodes in the tree */
int flag;

if((flag == ByPass(RTARROW_)) == TRUE)
  return(flag);

if(byPass(RTARROW_))
  newroot = CreateNode(RTARROW_);
newroot ->lptr = *root;
*root = newroot;
if(TypeExp(&((root)->rpcrf != TRUE)
  ErrorHandler(line_no,ERR9,(long)RTARROW_);
  return(ERROR_);
}
return(flag);

/***************************************************************************/

int TypeDom(root)
  NodeRec **root; /* root is a ptr to tree/subtree */
  /* currently working with */

/* <TYPEDOM> ::= <TYPETERM> (+ <TYPEDOM>) */
NodeRec *newroot; /* temp ptr to nodes in the tree */
int flag;

if((flag == TypeTerm(root)) == TRUE)
  return(flag);

if(byPass(ADD_))
  newroot = CreateNode(TYPEPLUS);
newroot ->lptr = *root;
*root = newroot;
if(TypeExp(&((root)->rpcrf != TRUE)
  ErrorHandler(line_no,ERR9,(long)RTARROW_);
  return(ERROR_);
int TypeTerm(root)
    /* root is a ptr to tree/subtree */
    NodeRec **root;
    /* currently working with */

/*
<TYPETERM> ::= <TYPEFAC>(*/ <TPERM>) *
*/
NodeRec *newroot;
/* temp ptr to nodes in the tree */
int flag;

if(flag = TypeFac(root)) == TRUE)
    if (ByPass(MULT_)) /* will recursively search for */
        newroot = CreateNode(TYPETIMES);
        newroot ->lptr = *root;
        *root = newroot;
        if(TypeTerm(!(*root) = rptr) == TRUE) /* fix root for return */
            ErrorHandler(line_no, ERR9, (long)MULT_);
            return(ERROR_);
    } /* end recursive search */

return(flag);
/* end TypeTerm() */

/*****************************************************************************/

int TypeFac(root)
    /* root is a ptr to tree/subtree */
    NodeRec **root;
    /* currently working with */

/*
<TYPEFAC> ::= <TYPEPRIMARY>? | <TYPEPRIMARY> |
/*
/*
<ID> '<<' <TYPEEXP> (,<TYPEEXP>)* '>>' <ACTUAL> */
/*
Where <<TYPEEXP,(TYPEEXP,...)>> and/or <ACTUAL> need not be present */
*/
NodeRec *newroot;
/* temp ptr to nodes in the tree */
int flag;
long ptr;

if(ptr = ByPass(IDENTIFIER_))
    *root = CreateNode(IDENTIFIER_);
    (*root) ->index = ptr;

if(ByPass(ST_SEQUENCE_) && ByPass(ST_SEQUENCE_))
    ErrorHandler(line_no, ERR7, NULL);
    return(ERROR_);
    /* end bypass << */
    goto CHECK;
    /* end if ID */

if(flag = TypePrimary(root)) == TRUE)
    goto CHECK;
    return(flag);
    /* return either ERROR or FALSE */
CHECK: if (ByPass(STAR_))
    { newroot = CreateNode(STAR_);
      newroot -> lptr = (*root);
      *root = newroot;
    } /* end if STAR */

    return(TRUE); /* made it this far, all OK */
} /* end TypeFac() */

******************************************************************************
int TypePrimary(root) /* root is a ptr to tree/subtree */
    NodeRec **root; /* currently working with */

    /* TYPEPRIMARY ::= <PRIMTYPE> | '(' TYPEEXP ')' */
    /* NOTE: ID already checked in TYPEFAC() */

    if (ByPass(LTPAREN_))
        if (TypeExp(root) != TRUE)
            ErrorHandler(line_no, ERR9, /* note it, no fix */
                          (Long)LTPAREN_);

        if (ByPass(RTPAREN_))
            return(TRUE);
        else
            ErrorHandler(line_no, ERR_f, /* long RTPAREN */
                          (Long)RTPAREN_);
            return(ERROR_);
    } /* end ByPass '(' */

    if (PrimType(root))
        return(TRUE);
    return(FALSE); /* default */

} /* end TypePrimary() */

******************************************************************************
int PrimType(root) /* root is a ptr to tree/subtree */
    NodeRec **root; /* currently working with */

    /* <PRIMTYPE> ::= real | integer | natural | boolean | trivial | type */

    if (ByPass(REAL_))
        { *root = CreateNode(REAL_);
          return(TRUE);
        } /* end if REAL */

    if (ByPass(INTEGER_))
    { *root = CreateNode(INTEGER_);
      return(TRUE);
    } /* end if INTEGER */

    if (ByPass(NATURAL_))
    { *root = CreateNode(NATURAL_);
      return(TRUE);
    } /* end if NATURAL */

    if (ByPass(BOOLEAN_))
    { *root = CreateNode(BOOLEAN_);

    */
return(TRUE);
}
/* end if BOOLEAN */

if(ByPass(TRIVIAL_))
{  *root = CreateNode(TRIVIAL_);
   return(TRUE);
}
/* end if TRIVIAL */

if(ByPass(KW_TYPE_))
{  *root = CreateNode(KW_TYPE_);
   return(TRUE);
}
/* end if TYPE */

return(FALSE);
/* default - none of the above */
/* end PrimType() */

**************************************************************************
PUBLIC DOMAIN SOFTWARE

Name:  Parser Utilities
File:  parsr_util.c
Authors:  Maj E.J. COLE / Capt J.E. CONNELL
Started:  01/26/87
Archived:  03/03/87
Modified:  04/23/87  FillBuffer() now calls GetToken() direct.

This file contains the utility modules for the parser:
- CreateNode()
- MakeNewRoot()
- ByPass()
- FillBuff()
- IsFormal()
- IBall()
- NodeName()
- EnterName()
- FindName()

Modified:  03/20/87 - Buffer Handling routines added - JC
- 04/23/87 - FillBuff() calls GetToken() direct vice working with intermediate file of tokens.
- EnterName() and FindName() added to place IDs, LITERALS, and CONSTANTS into the name table.  JC

#include <stdio.h>
#include <parser.h>

extern int line_no;            /* global var, holds line no 
extern FILE *pinfile;          /* of source prog    
extern FILE *wfile;            /* global working file 
char token[MAXLINE] = "x";     /* Init token[0] to value other 
NameRec *netable TABLESIZE - 1, /* length of the string. 
*EnterName();

 UTILITY

NodeRec *
CreateNode(op)
 NodeType op;
/* operator type of node */

/* Creates a tree node and returns the pointer (temp) to this node. */
/* Accepts node type (op), an integer, and inserts it into the node. */

NodeRec *temp;

/* create a node */

temp = CALLOC(1, NodeRec);
temp -> name = op;
temp -> ln = line_no;
temp -> lptr = (temp -> rptr) = NULL;
return(temp);

/* end CreateNode() */

MakeNewRoot(root, type, side)
NodeRec *newroot;

newroot = CreateNode(type);
(side == LEFT) ? (newroot ->lptr = *root) : (newroot ->rptr = *root);
*root = newroot;

/* end MakeNewRoot */

/*************************************************************************/

extern long tokenbuff[], *ptr;
int token_num;
NameRec *nptr;

ptr = start;

do

    token_num = GetToken(token);
    *ptr = token_num;
    ++ptr;

    switch (token_num)
    case LITERAL_:
        case CONSTANT_:
        case IDENTIFIER_:

        (token[0] = strlen(token));
        if((nptr=EnterName(token)))
            *ptr = (long)nptr;
            ++ptr;
        else ErrorHandler(NULL, ERR7, NULL);
        break;
    default:
        /* do nothing */
    end switch

while((token_num != EOF) && (ptr < &tokenbuff[BUFSIZE]));

ptr = &tokenbuff[0];
/* reset the buffer ptr */
/* end FillBuff() */

/*************************************************************************/
long
ByPass(tgt)
int tgt;
/* Checks to see if the next token in the buffer matches the target. */
/* If so, then returns the token no. and increments the buffer */
/* pointer */

extern long tokenbuff[], *ptr;
if(ptr >= &tokenbuff[BUFSIZE]) /* see if at end of buffer */
    FillBuff(&tokenbuff[0]); /* refill buffer */
while(*ptr == EOLN) /* increment counter & skip */
    ++ptr;
    ++line_no;
if(ptr >= &tokenbuff[BUFSIZE]) /* see if at end of buff */
    FillBuff(&tokenbuff[0]); /* refill buffer */
    /* end while */
if (*ptr != tgt) return(FALSE); /* otherwise, it was found */
if(ptr == &tokenbuff[BUFSIZE]) /* if at end of buffer */
    FillBuff(&tokenbuff[0]); /* refill buffer */
switch (tgt)
    case LITERAL : /* return ptr to struct */
        return(* (ptr++)); /* holding the token */
    case IDENTIFIER : /* just return true */
        return(tgt); /* end switch */
    case CONSTANT:
        return (tgt); /* end ByPass() */
    default:
        return(tgt); /* *end switch */

IsFormal(root)
int NodeRec *root; /* root is ptr to subtree */
/* currently working with */
/* Required to make the language deterministic. Compound() returned */
/* TRUE and "|->" was subsequently found. Formal is a proper subset of */
/* the compounds so need to insure no errors in the formals. */
/* Performs a preorder search of the subtree. NOTE: assumes that root */
/* initially points to a non-null compound list. */

#if defined DEBUG
printf("isformal entered, root->name = %d\n", root->name);
if (root == NULL) printf("root is null\n");
#endif
if (root == NULL)
    return(TRUE);

if(root->name==COMMA_ : root->name==IDENTIFIER_
    "root->name==ELLIST
}

103
if((IsFormal(root->iptr))
   && (IsFormal(root->rptr)))
return(TRUE);
/* end IsFormal */
/*********************************************************************************/

int IsAll(tgt, index)
int tgt, index;
/* Checks to see if the (index)th token in the buffer matches the */
/* target. If it does returns TRUE else FALSE. Does not increment */
/* the buffer pointer. Checks for full buffer implemented in this */
/* manner to allow for future flexibility. Could have used simple */
/* heuristic of: */
/* if(ptr + (3*index) > &tokenbuff[BUFSIZE]) RefillBuffer; */
/* at the expense of generality */
extern long tokenbuff[], *ptr;
long *tptr;
if(ptr >= &tokenbuff[BUFSIZE])
   FillBuff(tokenbuff[0]);
/* see if at end of buff if */
/* so, refill buffer */
DO_AGAIN:
   tptr = ptr;
while(*tptr == EOLN_)
   /* increment tptr & skip EOLNs */
   /* see if at end of buff */
   /* need to refill buffer and */
   /* then start over */
   /* end while */
   /* only enter for loop if need to */
   /* look more than one char ahead */
   /* double skip because next */
   /* entry is addr of element */
   for(index =1; --index)
      switch (*tptr)
         case IDENTIFIER:
            case CONSTANT:
            case LITERAL: tptr -= 2; break;
            case EOLN:
               while(*tptr == EOLN_)
                  /* increment counter & skip */
                  /* refill buffer & start over */
                  /* end while */
                  default: ++tptr;
                     /* end switch */
               if(tptr >= &tokenbuff[BUFSIZE])
                  goto REFILL;
                     /* check if will overflow buff */
               /* end for */
               if (*tptr == tgt) return(FALSE);
               else return(TRUE);
               /* take what's left in buffer, */
               /* put at beginning, now refill */
               /* rest of buffer */
               FillBuff(tptr);
               /* posit to end */
               REFILL:
               for(tptr = &tokenbuff[0];
                  ptr < &tokenbuff[BUFSIZE]; ptr++, tptr++)
                  /* refill buffer from current */
                  /* end for */
               /* end while */
               /* end DO_AGAIN */
               /* end IsAll */
               return(FALSE);
goto DO_AGAIN; /* refilled buffer, so start over */
/* end Ball() */

char *
NodeName(ptr)
    NodeRec *ptr;
    /* Accepts a ptr to a structure of NodeRec. Dereferences this node */
    /* to get a ptr to structure of NameRec which hold the string */
    /* containing the name of the value in NodeRec. Returns the name to */
    /* calling routine */

NameRec *temp;
    /* temp ptr to data struct holding name of *ptr */

    temp = (NameRec *)(ptr->index);
    return(temp->name + 1);
/* end NodeName() */

/*************************************************************************/
APPENDIX I
ROCK COMPILER — ERROR HANDLER

PUBLIC DOMAIN SOFTWARE

Name : Error Handler
File : errors.c
Authors : Maj E.J. COLE / Capt J.E. CONNELL
Started : 01/20/87
Archived : 04/07/87

This file contains the execution modules for error recovery.
ErrorHandler(), EatEm()

Algorithm : ErrorHandler() is called by other modules in the
compiler. It insures the error count is updated and
the error is written to the error file. If required,
ErrorHandler() calls EatEm() to gobble tokens to get to
a known point in the parse. Used during error
recovery. After MAXERRORS number of errors simply
returns to calling routine.

NOTE : 'errorfile' must have been initially created before
ErrorHandler() is first called - don't want to append
to last times errors!

Modified :

#include <stdio.h>
#include <scanner.h>
#include <errors.h>
extern FILE *errorfile; /* working file */
int num_errors = 0; /* running tally of # errors */
char *errors[] = {
0 /* incomplete "\n" ",
1 /* 2 /* "RESERVED FOR FUTURE USE",
2 /* 3 /* "\n" without following ", logical OR is " ",
3 /* 4 /* "S" without following "R", "N", "Z", or ":",
4 /* 5 /* "invalid numeric constant " <",
5 /* 6 /* "literal without ending "",
6 /* 7 /* "unidentified char in input program "",
7 /* 8 /* "MEMORY OVERFLOW DURING COMPILATION",
8 /* 9 /* "error in statement following "",
9 /* a /* "error in type definition following "",
10 /* b /* "unable to complete definition of blockbody after keyword LET",
11 /* c /* "missing or misplaced ";" after definition",
12 /* d /* "valid qualexp/exp not found in the def 'a'".};
"val type expr not found in the def",
"missing or error in format list",
"missing or missing",
"at least one identifier must follow keyword TYPE",
"unable to complete def after def following keyword AND",
"missing or missing after keyword WHERE",
"missing or missing part of format list",
"error in processing multiple Actuals",
"missing after keyword ELSE",
"missing or missing expr. with keyword THEN",
"keyword ELSE before keyword END",
"missing or missing expr. following IFWA expression",
"error in processing multiple Actuals",
"the feature can be implemented ...",

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......

......
execl("rock.exe", "rock.exe", NULL);

/* end if no more memory */

fprintf(errorfile, "line %d : %s ",
        line_no, errorslerror(err_no));

switch (err_no) {
    case ERR4:
        case ERR5: fprintf(errorfile, "%s
", (char *)str_num); break;
        case ERR6: fprintf(errorfile, "%s
", (char *)str_num); break;
        case ERR8: switch(str_num)
            case LEQ : fprintf(errorfile, "<=\n"); break;
            case NEQ : fprintf(errorfile, "<>\n"); break;
            case EQ : fprintf(errorfile, "\n"); break;
            case ADD : fprintf(errorfile, "+\n"); break;
            case SUB : fprintf(errorfile, "-\n"); break;
            case SUBSCRIPT : fprintf(errorfile, "\n"); break;
            case COLON : fprintf(errorfile, ":\n"); break;
            case CAT : fprintf(errorfile, "\n"); break;
            case LINETARROW : fprintf(errorfile, "->\n"); break;
            case (KW_LESS_): fprintf(errorfile, "LESS\n"); break;
            case (KW_GREATERN_): fprintf(errorfile, "GREATER\n"); break;
            case (KW_EQUAL_): fprintf(errorfile, "=\n"); break;
            case (KW_INSERT_): fprintf(errorfile, "\n"); break;
            case (KW_ADD_): fprintf(errorfile, "+\n"); break;
            case (KW_MULTIPLY_): fprintf(errorfile, "\n"); break;
            case (KW_SUBTRACT_): fprintf(errorfile, "-\n"); break;
            case (KW_DIVIDE_): fprintf(errorfile, "\n"); break;
            case (KW_ASSIGN_): fprintf(errorfile, "\n"); break;
            case (KW_BITWISE_AND_): fprintf(errorfile, "\n"); break;
            case (KW_BITWISE_XOR_): fprintf(errorfile, "\n"); break;
            case (KW_BITWISE_OR_): fprintf(errorfile, "\n"); break;
            case (KW_BITWISE_NOT_): fprintf(errorfile, "\n"); break;
            case (KW_COLON_): fprintf(errorfile, ":\n"); break;
            case (KW_CAT_): fprintf(errorfile, "\n"); break;
            case (KW_LSP_): fprintf(errorfile, "\n"); break;
            case (KW_RSP_): fprintf(errorfile, "\n"); break;
            case (KW_SEQ_): fprintf(errorfile, "\n"); break;
            case (KW России): fprintf(errorfile, "\n"); break;
            case (KW_END_): fprintf(errorfile, "\n"); break;
            default:
                fprintf(errorfile, "UNDEFINED error\n");
                break;
        }

    case ERR9: switch(str_num)
        case ADD : fprintf(errorfile, "+\n"); break;
        case MULT : fprintf(errorfile, "\n"); break;
        case RTARROW : fprintf(errorfile, "->\n"); break;
        case LPAREN : fprintf(errorfile, "\n"); break;
        case RPAREN : fprintf(errorfile, "\n"); break;
        default:
            fprintf(errorfile, "UNDEFINED error\n");
            break;

    case ERR10:
        switch(str_num)
            case KW_\&_AND_:
                break;
            case KW_\&_WHERE_:
                fprintf(errorfile, "\n");
                break;
            case RTARROW_:
                fprintf(errorfile, "->\n");
                break;
            case LPAREN_:
                fprintf(errorfile, "\n");
                break;
            case RPAREN_:
                fprintf(errorfile, "\n");
                break;
            default:
                fprintf(errorfile, "UNDEFINED error\n");
                break;

    case ERR11:
        switch(str_num)
            case RTARROW_:
                str_num=NULL; break;
            case LPAREN_:
                str_num=NULL; break;
            case RPAREN_:
                str_num=NULL; break;
            case END_SEQUENCE_:
                fprintf(errorfile,">\n");
                str_num=NULL; break;

108
case KW-END:
    fprintf(stderrfile,"KEYWORD END\n")
    str_num = KW; break; /* set up for call to EatE() */
case KW-THEN:
    fprintf(stderrfile,"KEYWORD THEN\n")
    break;

default:
    fprintf(stderrfile,"UNDEFINED error\n")
    break; /* end switch case ERR_ */
case ERR_:
    switch(str_num)
        case IF_:
            fprintf(stderrfile,"IF\n")
            break;
        case ELIF_:
            fprintf(stderrfile,"ELIF\n")
            break;
        case ELSE_:
            fprintf(stderrfile,"ELSE\n")
            break;
        case THEN_:
            fprintf(stderrfile,"THEN\n")
            break;
        case BEGIN_:
            fprintf(stderrfile,"BEGIN\n")
            break:
        default:
            fprintf(stderrfile,"UNDEFINED error\n")
            /* end switch case ERR_ */
    str_num = KW;
    break; /* set str_num up to be passed to EatE() */
default: fprintf(stderrfile,"\n")
    break; /* end switch */

if (err_no > ERR_a) &&
    (err_no < ERR_a) &&
    (str_num = NULL) &&
    EatE((int)str_num); /* end ErrorHandler */

/****************************************************************************

void
EatE(tgt) int tgt;

/* Increments token buffer pointer until tgt token is found. */
/* Use in error recovery to reach a known point in the program. */
extern long tokenbuf**, *ptr;
extern int line_no;

#define DEBUG
printf "eaten entered, tgt = \%d", tgt; *errf

while(*ptr != EOF_)
    switch (tgt) 
        case EOLN : /*ptr++; line_no++; break;
        case SEMI_ : if (*ptr == SEMI) (*ptr++) =; */
return;
--ptr; break;

case EQUIV: switch ((int)*ptr)
  case EQUIV:
    case SEMI:
      case KW_AND:
        case KW_LET:
          default: ++ptr;
          break; /* end switch case EQUIV */
    case KW_WHERE:
      switch ((int)*ptr)
        case KW_WHERE:
          case KW_AND:
            case KW_LET:
              case SEMI:
                return;
                default: ++ptr;
            break; /* end switch case WHERE */
        case KW_AND:
          switch ((int)*ptr)
            case KW_AND:
              case KW_LET:
                case SEMI:
                  return;
                  default: ++ptr;
            break; /* end switch case AND */
      case RTPAREN:
        switch ((int)*ptr)
          case RTPAREN:
            case LTPAREN:
              case RTPAREN:
                case COMMA:
                  case EQUIV:
                    case LINTERARROW:
                      case KW_LET:
                        case KW_AND:
                          case SEMI:
                            return;
                            default: ++ptr;
                        break; /* end switch case RTPAREN */
            case KW_IF:
              case KW_ELSE:
                case KW_THEN:
                  switch ((int)*ptr)
                    case KW_ELSE:
                      case KW_ENDIF:
                        case KW_THEN:
                          return;
                          default: ++ptr;
                    break; /* end switch case THEN, etc */
      case COMMA:
        switch ((int)*ptr)
          case COMMA:
            case LTPAREN:
              case RTPAREN:
                case LTSQUIG:
                  case RTSQUIG:
                    case STSEQUENCE:
                      case ENDSEQUENCE:
                        case SEMI:
                          case KW_LET:
                            case KW_WHERE:
                              case KW_AND:
                                return;
                                default: ++ptr;

break; /* end switch case COMMA */

case KW_-END_ :
    break;

case KW_-BEGIN_ : switch ((int)*ptr); 
    case KW_-END_ :
    case KW_-LET_ :
    case KW_-WHERE_ :
    case KW_-AND_ :
    case COMMA_ : 
    case RTPAREN_ : 
    case RTSQUIG_ : 
    case END_SEQUENCE_ : 
    case SEMI_ : return;
    default : **ptr;
    break; /* end switch case BEGIN/END */

default : 
    return; /* end switch */

} /* end while */

} /* end EatEm() */

*******************************************************************************
APPENDIX J
ROCK COMPILER — SEMANTIC CHECKER

/* PUBLIC DOMAIN SOFTWARE */
/* */
/* Name : Semantic Checker Module 0 */
/* File : SemO.c */
/* Authors : Maj E.J. COLE / Capt J.E. CONNELL */
/* Started : 02/01/87 */
/* Archived : 04/03/87 */
/* Modified : */

*******************************************************************************
** This file contains the following modules for the PHI parser: **
** Hnumconvert Numconvert **
** Algorithm : **
** This module contains procedures for type conversion. If the **
** rt child of a node may be converted to the lt type but the con- **
** verse is not true, "Hnumconvert" is called. If either side may be **
** converted, "numberconvert" is called **
*******************************************************************************
** Modified : **
*******************************************************************************

/*******************************************************************************
** Externals ***************************************************************************/
#include <semcheck.h>

extern void terror ();

Phitype
hnumconvert (ltype, rtype, ptr) Phitype

/* Type conversions for the */
/* right side of the tree only */
/* Left and Right types */
/* Ptr to the root working with */
/* Generates code to convert */
/* integer/natural to real */

if (((ltype == BOOLEAN) & (rtype == BOOLEAN))
return (BOOLEAN);

switch (ltype) {
    case (REAL) : switch (rtype) { /* Predicate actions on type of lt */
        case (REAL) : return (REAL);
        case (INTEGER) :
        case (NATURAL) :
            c_ztor ();
            return (REAL);
        default : /* Generate code for conversion */
            */ side of node */
            /* Matching types; no conv req */
        default : /* Generate code for conversion */
            */ side of node */
            /* Matching types; no conv req */
    }
}

112
terror (ERR aa, ptr->in); /* No appropriate match: error */
return (REAL); /* Ret real so semantic check cont */

case (INTEGER) : switch (rtype) {
  case (INTEGER) :
  case (NATURAL) : return (rtype);
  default : 
    terror (ERR_co, ptr->ln);
    return (INTEGER); }

case (NATURAL) : 
  if (rtype == NATURAL) 
    return (rtype);
  else 
    terror (ERR_bb, ptr->ln);
    return (NATURAL);
  
default : terror (ERR_aa, ptr->ln);
    return (NATURAL);

} /* Numconvert */

PHITYPE
numconvert (ptr) /* Do number conversions for */
  nodal ptr;
(PHITYPE ltype, rtype;
  extern PHITYPE semcheck ();
  extern void c_ztor ();

  ltype = semcheck (ptr->:ptr); /* Get left type */

  if (ptr->rptr->name == (KW - ENDIF_)) /* Special case of "if" sequence */
    return (ltype);

  rtype = semcheck (ptr->rptr); /* Get right type */

  if (!(ltype == BOOLEAN) && (rtype == BOOLEAN)) /* No conversion necessary */
    return (BOOLEAN);

switch (ltype) { /* Predicate actions on ltype */
  case (REAL) : switch (rtype) { /* Types are same: no action req */
    case (REAL) : return (REAL);
    case (INTEGER) :
    case (NATURAL) :
      c_ztor ();
      return (REAL);
    default : 
      terror (ERR_aa, ptr->rptr->ln);
      return (REAL);
    }

  case (NATURAL) : switch (rtype) : /* Convert left side */
    case (REAL) :
      c_ztor ();
      return (REAL);
    case (INTEGER) :
    case (NATURAL) :
      return (INTEGER);
    default : 
      terror (ERR_aa, ptr->rptr->ln);
      return (NATURAL);

  case (INTEGER) : switch (rtype) : /* Convert right side */
    case (REAL) :
      c_ztor ();
      return (REAL);
    case (INTEGER) :
    case (NATURAL) :
      return (INTEGER);
    default : 
      terror (ERR_aa, ptr->rptr->ln);
      return (NATURAL);

  case (BOOLEAN) : switch (rtype) { /* Types are different: action req */
    case (REAL) : return (REAL);
    case (INTEGER) :
    case (NATURAL) :
      return (INTEGER);
    default : 
      terror (ERR_aa, ptr->rptr->ln);
      return (INTEGER);
    }

  default : 
    terror (ERR_bb, ptr->ln);
    return (INTEGER);

} /* Numconvert */
case (INTEGER) : switch (itype) {
    case (REAL) : c_ztor (); return (REAL); /* Convert left side */
    case (INTEGER) :
    case (NATURAL) : return (INTEGER); /* No conversion necessary */
    default : terror (ERR_aa, ptr->rptr->in); return (NATURAL);
}

default : terror (ERR_aa, ptr->lptr->in); /* Types are not numeric */
    return (NATURAL);
PUBLIC DOMAIN SOFTWARE

Name: Semcheck Module 1
File: Seml.c
Authors: Maj E.J. COLE / Capt J.E. CONNELL
Started: 01/02/87
Archived: 01/10/87
Modified:

This file contains the following modules for the PHI parser:

Tletdef Trtarrow Tkindef
TWhere Tdataauxdef Tauxand
TAndcheck Tauxand Ttypetimes

Algorithm:
This module contains scoping procedures (TWhere and Tauxand)
definition procedures (trtarrow, tkindef, ttypetimes) and the data
definition procedure.

Modified:

externals

---

```c
#include <semcheck.h>
#include <string.h>
extern int typeptr; /* For "strcpy"
extern node types ();
extern void terror ();

tnode *fhead = NULL;

void tletdef (ptr)
  /* checks types of both branches */
  node *ptr;
  semcheck (ptr->lptr);
  semcheck (ptr->rptr);

void trtarrow (ptr)
  /* Returns type */
  node *ptr;
  PHITYPE ttype, rtype;
  extern void putform ();
  ttype = semcheck (ptr->lptr);
  rtype = semcheck (ptr->rptr);
  if ((ptr->lptr->name == TYPETIMES)
      || (ptr->rptr->name == TYPEPLUS))
    putform (ttype);
  return (rtype);
```

---

115
void tkindef (ptr)
    /* Adds variable name to defstack */
    nodal ptr;
    (extern defptr defhead;
    extern void putdef ();
    PHITYPE type;
        rtype = semcheck (ptr->rptr);
    putdef (rtype, ptr->lptr);
    defhead->fptr = fhead;
    fhead = NULL;
    */

/**************************** Twhere ***************/
PHITYPE twhere (ptr)
    /* Semcheck where node */
    nodal ptr;
    PHITYPE type;
        semchecker (ptr->lptr);
    type = semchecker (ptr->rptr);
    return (type);
    */

/**************************** TDatauxdef *************/
void tdatauxdef (ptr)
    /* WORKS FOR ONE FORMALS ONLY */
    nodal ptr;
    (extern void c_store_code (), c_jmp ();
    extern PHITYPE getdtype ();
    extern defptr finddef ();
    extern char *name ();
    defptr d_ptr;
    char *holder = malloc (8),
        *name = malloc (8);
    PHITYPE rtype,
    type,
    count = 0;
    name = strcpy (name, name());
    c_jmp (name);
    holder = strcpy (holder, name());
    c_start_proc (holder);
    rtype = semcheck (ptr->rptr);
    if (ptr->lptr->name == IDENTIFIER) {
        if ((d_ptr= finddef (ptr->lptr->index)) != NULL) {
            ptr->lptr->type = rtype;
            putvar (rtype, ptr->lptr);
        }
        else if (d_ptr->fptr == NULL) {
            ptr->lptr->type = getdtype (d_ptr);
            type = hnumconvert (ptr->lptr->type, rtype, ptr);
            putvar (type, ptr->lptr);
        }
        else error (ERR_dd, ptr->lptr->n);
    } else if (d_ptr->fptr == NULL) {
        ptr->lptr->type = getdtype (d_ptr);
        type = hnumconvert (ptr->lptr->type, rtype, ptr);
        putvar (type, ptr->lptr);
    } else error (ERR_dd, ptr->lptr->n);
    */
while (*holder - count) != NULL) {
    // Push piano through the door to copy strings
    (ptr->lpctr->label | count) = (*holder - count);
    --count;
}

// Generate code to end procedure
/* CANNOT USE C_ENDPROC() HERE; */
/* NO SCOPE CHANGE */
c_store_code ("ret\n");
c_store_code ("mne");
c_store_code (";\n");

/********************** And_Check **********************/
void
and_check (mark, ptr, mark_and)
varptr mark;
    and_ptr *mark_and, ptr;
extern varptr varhead;
extern int buff_ptr;
extern char *code_buffer;
int buff_holder;
varptr v_ptr = varhead;

if (ptr != NULL) {
    and_check (mark, ptr->link, mark_and);
    do {
        // Ptr = NULL is base for recur
        if (v_ptr->lptr->index==ptr->lptr->index)
            buff_holder = buff_ptr;
        buff_ptr = ptr->buff_ptr;
        c_call_proc (v_ptr->lptr->label);
        buff_ptr = buff_holder;

        if (*mark_and == ptr)
            *mark_and = ptr->link;
        del_and (ptr);
        break;
    }

    if (v_ptr == mark) break;
}
while (TRUE);

/********************** Tauxand **********************/
void
tauxand (ptr)
   egal_ptr;
    extern FLAG and_flag;
extern and_ptr and_head;
int save_and;
varptr mark;
and_ptr lptr, mark_and = and_head;

save_and = and_flag;
and_flag = TRUE;

/* Semantic check for and node */
/* Holder for and flag */
/* Mark top entry in var_list */
/* Mark current head of and_stack */
/* Save current and_flag */
/* Set and_flag */
semcheck (ptr->i ptr); /* Semantic Check */
mark = varhead;
semcheck (ptr->rptr);
and_check (mark, and_head, &mark_and); /* Check all new fctn & data cefs */
and_flag = save_and; /* Restore and flag */
tptr = and_head;
while (tptr != NULL) /* Traverse list until end */
    tptr = tptr->link;
if (mark_and != and_head) /* Undefine variables found */
terror (ERR_ee, ptr->in);

TTypeTimes

PHITYPE
ttypetimes (ptr) /* Semantic check "*" when used */
    nodal ptr;
    (extern void putform ());
    PHITYPE type;
    putform (semcheck (ptr->i ptr)); /* Attach formal type to */
    /* formal list */
    if (type = semcheck (ptr->rptr)) /* Look for right type; if 0, */
        putform (type); /* end of insertions */
    return (NULL); /* Always return NULL; */
    /* This value is used by parent */
PUBLIC DOMAIN SOFTWARE

Name: Semcheck Module 2
File: Sem2.c
Authors: Maj E.J. COLE / Capt J.E. CONNELL
Started: 01/02/87
Archived: 04/10/87
Modified:

This file contains the following modules for the PHI parser:

- Matchfor
- Tfunauxdef
- Tfunid
- Tactualist
- Tid
- Telist

Algorithm:

This module contains the procedures needed to define and call functions. Tfunauxdef will set up the run-time structure of the function, Tfunid will check the semantics of the function & matchfor, called by tfunid, checks for the proper type & number of formal parameters.

Tactualist coordinates the checking of a function call. It uses both telist and act_walk. Actwalk determines whether the number & type of actuals is correct, and telist checks each element list and returns its type.

Tid performs semantic checking for program variables.

Modified:

**************************************************************************** Externals ****************************
#include <semcheck.h>
#include <string.h> /* For "strcpy"

extern node types ();
extern varptr varhead;
extern void terror (); c_store_code ();

**************************************************************************** Globals ****************************
int actual_count = 0; /* count of all actuals

**************************************************************************** Matchfor ****************************

FLAG
matchfor (tptr, def)

formal nptr;
defptr def;

extern long curr_addr;
extern fnode *getfptr ();
extern FLAG form;

fnode *tptr = getfptr (def);
form = FLAG;
tptr = def->fptr;
curr_addr = 0;
if (nptr->name == IDENTIFIER) { /* Only one formal */ 
    (nptr->type) = (tptr->type); 
    nptr->addr = curr_addr; 
    putvar (tptr->type, nptr); 
    nptr = nptr->rptr; 
    tptr = tptr->link; 
} 

else { /* Multiple forms */ 

do { 
    nptr->iptr->type = tptr->type; 
    nptr->iptr->addr = curr_addr; 
    curr_addr = curr_addr - 
        types[t.ptr->type].bytes; 
    putvar (tptr->type, nptr->iptr); 
    nptr = nptr->rptr; 
    tptr = tptr->link; 
} while ((nptr != NULL) && (tptr != NULL)); /* Halt when end reached */ /* by either ptr */ 

    form = FALSE; 

    if (nptr == NULL && tptr == NULL) /* One ptr isn't at end of tree */ 
        return (FALSE); /* Error handled in calling form */ 

    else return (TRUE); 

} /********************* Ifunauxdef *********************/ 

void 
ifunauxdef (ptr) 
    nodal_ptr; 
    /* Type check ifunauxdef */ 

    extern long curr_addr; 
    extern void c_end_proc (), c JMP (); 
    extern char *name (); 
    extern nodal num_convert (); 
    char *the = malloc (8); 
    SHRTYPE _type, rtype; 
    varptr varl, mark = varhead; 
    curr pres_addr = curr_addr; 

    the = strcpy (name, name()); /* Name for JMP around function */ 
    JMP (the); /* Gen code to jump around func */ 

    _type = sem_check (iptr->type); 
    rtype = sem_check (iptr->rptr); 

    while (varhead->link != mark) { /* Eliminate forms from link list */ 
        varl = varhead; 
        varhead = varhead->link; 
        varl->link = NULL; 
        free (varl); 
    }

    tptr->rptr = 
        num_convert (_type, rtype, tptr->rptr); /* Convert if needed */ 
    c_end proc (the); 
    curr_addr = pres_addr; /* Reset addresses */
/***** Tfunid *************/

TFUNID

tfun (ptr)

/* Semantic Check for tfun */

void

extern def *fnddef ();

/* Generic Loop variant */

extern long data_addr;

int count = 0;

def ptr slist;

/* Func name not found */

if (!fnddef (ptr->lptr->index))

/* Match forms */

else if (!strcmp (ptr->lptr->name, dname))

/* Push plane -> door to copy */

/* string to array */

else

/* Gen code for begin function */

return (ptr->type);

/* semantic check for elt-ent li */

Tellist

void

Tellist (ptr)

/* Generate code */

if (ptr->lptr != NULL)

set (ptr->lptr);

set (ptr->lptr);

store_code ("call " ptr->lptr);

store_code ("push eax");

store_code ("push eax");

/* actual_count */

Act_Walk

void

act_walk (ptr, fptr)

/* recursive procedure */

set (fptr->actual_count);
nodal ptr;
fnode *fptr;

if (ptr->rptr != NULL) { /* Recurse until NULL ptr is hit */
    act_walk (ptr->rptr, fptr->link);
}

semcheck (ptr->lptr);
if (ptr->lptr->name == ELLIST) { /* Inc count only if left */
    ++actual_count;
    c_store_code ("call ppop\n");
    c_store_code ("push cx\n");
    c_store_code ("push dx\n");
}

/********************* Tactuals **********************/

PHITYPE

actuals (ptr) /* Evaluate actuals */
    nodal ptr;
    (extern void c_call_proc();
    extern FLAG and_flag;
    extern varptr findvar();
    extern defptr finddef();
    extern char *name();
    defptr def = finddef (ptr->lptr->index); /* Defstack pointer */
    varptr var = findvar (ptr->lptr->index); /* Varstack pointer */
    int count_hold = actual_count; /* Buffer for long to string conv */
    long convert; /* Conversion variable */
    fnode *fptr;
    actual_count = 0;
    fptr = def->fptr; /* Get a ptr to the formal nodes */
    act_walk (ptr->rptr, fptr);
    convert = actual_count; /* Generate code to put # of */
    c_store_code ("mov bx, "); /* actuals on the stack */
    stcl_d (long_buff, convert);
    c_store_code (long_buff);
    c_store_code ("\n");
    c_call_proc ("i_mov");
    stcl_d (long_buff, convert);
    c_store_code (long_buff);
    c_store_code ("\n");
    c_call_proc ("i_mov");
    if (((!var && !and_flag) || var)) /* Cover "and" scoping rules */
        add_and (ptr->lptr);
        c_call_proc (name ()); /* Holder for real name */
        else
            c_call_proc (var->nptr->label); /* Gen code to call function */
            actual_count = count_hold; /* Restore actual count */
            return (def->type);
        }
    terror (ERR_nh, ptr->ln);
    return (NOTFOUND);
}
/* PHCYPRE */
tid (ptr):
  node ptr;
  extern void s_u_form ();
  extern long buff_addr;
  extern char *name ();
  extern int format ();
  extern FLAG and_flag;
  extern varptr (varptr);
  extern setptr (setptr);
  char *long_buff = malloc (1);
  varptr var = finavar (ptr->pfix);
  setptr def;

  if (var) {
    if (def->fndef (ptr->apfix)) {
      add and (ptr);
      c_call_proc (name ...);
      return (gettype def);
    } else return (NOTFOUND);
  } else if (format (var) &&
    std_1ong_buff, var->aptr->aptr;)
    s_u_form (long_buff);
  else
c_call_proc (var->aptr);aptr;
return (settype (var));
PUBLIC DOMAIN SOFTWARE

Name : Semcheck Module #3
File : Sem3.c
Authors : Maj E.J. COLE / Capt J.E. CONNELL
Started : 01/02/87
Archived : 04/02/87
Modified : 

This file contains the following modules for the PHI parser:

- Trdivide
- Tidivide
- Tarithop
- Tprimary
- Tconvert
- Tconstant
- Tand
- Tor
- Tnegation

Algorithm:

This module contains the procedures necessary for implementing arithmetic & boolean operators. Tarithop coordinates the semantic checking of arithmetic ops by calling the proper function based on the operator type. Trdivide & Tidivide handle semantic checking for real & int division, respectively. For all other arithmetic ops, the numconvert procedure (sem0) is called to perform semantic checking, then code is generated.

For each boolean operator, the appropriate child(ren) is checked and code is generated for the operation.

For each boolean operator, the appropriate child(ren) is checked and code is generated for the operation.

Tconstant checks the type of a simple constant by calling convert, & then returns either the constant type or an error.

Modified:

/**************************** Externals *******************************

```
#include <semcheck.h>
#include <string.h> /* For "strlen()"

extern void terror ();
extern void c_store_code (); /* Store asm language output to a buffer
```

/**************************** Trdivide *****************************/

```
void trdivide (ptr)
   type, rtype;
   RTYPE (type, rtype);
   ext- FLAG err_found;
extern void c_store_code ();

    .type = semcheck (ptr->lptr); /* Check left side for type

    switch (type):
    case (REAL) : break;
    case (INTEGER) :
    case (NATURAL) :
    default : terror (ERR_aa, ptr->.ptr->ln); /* Lt child must return numeric type
                retur;
```

124
rtype = semcheck (ptr->rptr); /* Check right side for type */

switch (rtype) {
    case (REAL) : break;
    case (INTEGER) :
    case (NATURAL) :
        c_ftor ();
        break;
    default : terror (ERR_aa, ptr->rptr->ln);
               return; /* Error, no need to go thru acode */
}

acode (ptr, REAL); /* Generate code */

/*-------------------------------------------------------------------------*/
/*
** PHITYPE
**
** tidivide (ptr) /* Semcheck for integer division */
**
** nodal ptr;
**
** (PHITYPE ltype, rtype, type = NATURAL):
**
** ltype = semcheck (ptr->lptr); /* TypeCheck both sides */
** rtype = semcheck (ptr->rptr);
**
** switch (ltype) {
** case (INTEGER) : type = INTEGER;
** case (NATURAL) : break;
** default : terror (ERR_cc, ptr->lptr->ln);
**             return (INTEGER);
**     }
**
** switch (rtype) {
** case (INTEGER) : type = INTEGER;
** case (NATURAL) : break;
** default : terror (ERR_cc, ptr->rptr->ln);
**             return (INTEGER);
**     }
**
** acode (ptr, type); /* Generate code */
**
** return (type);
**}

/*-------------------------------------------------------------------------*/
/*
** PHITYPE
**
** arithop (ptr) /* Type Check Addition, Multiplication, Sequence Ops */
**
** nodal ptr;
**
** (extern PHITYPE numconvert ();
**
** int type;
**
** switch (ptr->name) {
** case (ADD_) : /* Addition fails through */
**     }
** case (SUB_) : /* Subtraction fails through */
**     }
** case (MULT_) : if (ltype = numconvert (ptr)) {
**        acode (ptr, type);
**        return (type);
**    }
** else {
**        terror (ERR_aa, ptr->ln);
**        return (NATURAL);
**    }
**
** return (type);
**}

125
case (RDIV_) : tdivide (ptr);
ptr->type = type;
return (REAL);
case (IDIV_) : tdivide (ptr);
ptr->type = type;
return (INTEGER);
case (COLON_) : break; /* Dummies for now, */
/* but watch our smoke: */
case (CAT_) : break;
}

/**************************** Tprimary *****************************/

PHITYPE
tprimary (ptr) /
  nodal ptr;
  (PHITYPE type;
    type = semcheck (ptr->rptr);
    if ((type != INTEGER) &&
        (type != REAL) &&
        (type != NATURAL)) /* Check type of right node */
      error (ERR_aa, ptr->rptr->in); /* Type must be a number */
    else if ((ptr->name) == NEG_) /* Negate operation */
      c_store_code ("call igetvalue\n"); /* Spew code */
      c_store_code ("call iputvalue\n"); /* Note that no action is req */
      c_store_code ("call iputvalue\n"); /* for unary "+" */
    return (type);
  }

/**************************** Convert *****************************/

PHITYPE
convert (string) /* Convert const to real, boolean,"*/
  stg string;
  (FLAG e = FALSE,
    period = FALSE;
    int count = 0;
    if ((strcmp (string, "FALSE")
        && strcmp (string, "TRUE"))) /* If not boolean */
      if ((string [count] != 0) { /* Loop until end of string */
        if (!isdigit (string [count])) /* If character is not a digit */
          if ((string [count] == 'e') ||
              (string [count] == 'E')) /* "e" or "E" found */
            if (e) return (ERROR); /* Cannot have two "e"s */
            e = TRUE;
            if ((string [count + 1] == '+') ||
                (string [count + 1] == '-')) /* "+" or "-" character */
              ++count;
}
else if (string['count'] == '.'); // Decimal point found
  // Cannot have two periods
else period = TRUE;
else return (ERROR);

--count;

if (e period) return (REAL);
/* If exponent has been r.n.,
period or "e" makes real
* Negative sign makes an integer
return (INTEGER);
* If no other num types, natural
return (NATURAL);
* If not a number, a boolean
return (ERROR);
*/

/*******************************************************************************
PHYPE
tconstant (ptr)
  * Handle constant nodes
  nodal ptr;
  extern put_addr ();
  PHYPE type;
  NameRec *tptr;
  tptr = ptr->index;
  if (type = convert (tptr->name + 1)) {
    ptr->type = type;
    put_addr (ptr, type);
    c_lconst (tptr->name + 1);
    return (type);
  }
  terror (ERR_z, ptr->In);
/*******************************************************************************
PHYPE
tand (ptr)
  * Semantic check for boolean node
  nodal ptr;
  PHYPE ltype, rtype;
  ltype = semcheck (ptr->lptr);
  rtype = semcheck (ptr->rptr);
  if ((ltype == BOOLEAN && rtype == BOOLEAN)) /* Both children must be relative */
    terror (ERR_kk, ptr->ln);
  c_store_code ("call land\n");
  return (BOOLEAN);
}*/

/*******************************************************************************
PHYPE
tor (ptr)
  * Semantic check for or node
  nodal ptr;
  PHYPE ltype, rtype;
  ltype = semcheck (ptr->lptr);
  rtype = semcheck (ptr->rptr);
  if ((ltype == BOOLEAN && rtype == BOOLEAN)) /* Both children must be relative */
  terror (ERR_kk, ptr->ln);

127
c_store_code ("call xor\n"); // Generate code

return (BOOLEAN);

/**************************** Tnegation **********************/

PHYPE

tnegation (ptr)

node ptr;

if (!secheck (ptr->rptr == BOOLEAN))
    // It child must be a boolean;
    terror (ERR_xx, ptr->ln);
else c_store_code ("call negation\n"); // Gen code for boolean negation

return (BOOLEAN);
This file contains the following modules for the PHI compiler:

- **Tif**  
  - **Then**  
  - **Telseif**  
  - **Telse**  
  - **Tcomp**

**Algorithm:**

This module contains the procedures necessary to implement the "if-then-elseif-else" series of commands. Tif coordinates the semantic checking by calling Then to check its left nodes, then calling Telse to check its right nodes. Telse will be called until the right subtree runs out of "elses" and "elseifs".

**Modified:**

```
#include <semcheck.h>
#include <string.h>
extern FLAG err_found;
extern PHITYPE semcheck ();
extern char *name ();
extern void terror (), c_store_char ();

char *if_label = NULL;

PHITYPE
Tif (ptr)
   nodal ptr;
   extern PHITYPE numconvert ();
   PHITYPE type;
   if (if_label == NULL) if_label = malloc (8);
   if_label = strcpy (if_label, name ());
   type = numconvert (ptr);
   if err_found == 1 { c_store_code (if_label); }
   else { c_store_code ("\n\n");
     return (type);
   }
```
/******************************************************** Tthen ********************************************************/ PHITYPE tthen (ptr) /* Sem checker for then node */ nodal ptr; /* Pointer to the node */ (PHITYPE ltype, rtype; /* Type returned from left */ cntr *label = calloc (7,1); /* Jump for assembly code */ cntr *holder = calloc (7,1); strcpy (holder,if_label); if(ltype==semcheck (ptr->lptr) != BOOLEAN) /* Left node contains condition; */ terror (ERR_11, ptr->lptr->ln); if_label = strcpy (if_label,holder); label = strcat (label, name()); /* Get a label for assembly code */ c_store_code ("call qgetvalue\n"); c_store_code ("cmp ax,0\n"); c_store_code ("jne "); c_store_code (if_label); c_store_code ("\n"); rtype = semcheck (ptr->rptr); /* Check right side */ c_store_code ("jmp "); /* Generate code */ c_store_code (if_label); c_store_code ("\n"); c_store_code (if_label); c_store_code (":\n"); return (rtype); /* Right type is returned */ 

/******************************************************** Telseif ********************************************************/ PHITYPE telseif (ptr) /* Sem check for "elseif" node */ nodal ptr; /* Ptr to the node */ extern PHITYPE numconvert (); /* Function converts and returns */ return (numconvert (ptr)); 

/******************************************************** Telse ********************************************************/ PHITYPE telse (ptr) /* Sema checker for "else" node */ nodal ptr; /* Node */ return (semcheck (ptr->lptr)); /* Return left side; */ /* right side is always empty */ 

/******************************************************** Tcomp ********************************************************/ PHITYPE tcomp (ptr) /* Handle comparisons and */ nodal ptr; /* set membership operations */ (extern PHITYPE numconvert (); PHITYPE type; 

130
```c
/* Check and convert if necessary */
/* THIS IS FOR FUTURE USE WHEN */
/* REALS ARE IMPLEMENTED */
/* Check cases */
/* WORKS ONLY FOR INTEGERS AND */
/* BOOLEANS --- NEEDS REAL */

switch (ptr->name) {
    case (EQ) : c_store_code ("call lequ\n");
                 break;
    case (NEQ) : c_store_code ("call ineq\n");
                 break;
    case (KW_ * LESS_) :
                 c_store_code ("call llt\n");
                 break;
    case (KW_ * GREATER_) :
                 c_store_code ("call lgt\n");
                 break;
    case (LEQ) : c_store_code ("call liteq\n");
                 break;
    case (GEQ) : c_store_code ("call lgeq\n");
                 break;
    case (KW_ * IN_) :
                 c_store_code ("call lin\n");
                 break;
    case (KW_ * NOTIN_) :
                 c_store_code ("callnotin\n");
                 break;
    default : terror (ERR_11, ptr->ln);
                 break;
}
return (BOOLEAN);
```

PUBLIC DOMAIN SOFTWARE

Name: Semcheck Utilities.l
File: Sem_U.c
Authors: Maj E.J. COLE / Capt J.E. CONNELL
Started: 01/02/87
Archived: 04/03/87

This file contains the following modules for the PHI parser:

- Putvar
- Putform
- Makeform
- Findvar
- Getfptr
- Getvtype
- Finddef
- Put_addr
- Name
- Getdtype
- Form
- Makevar
- Putdef
- AndAlloc
- Add_And
- Del_And

******************** Externals **********************
#include <semcheck.h>
#include <string.h>

Flags
err_found = FALSE;
long curr_addr = START_ADDR;
long curr_scope = START_ADDR;
form = FALSE;

Typetable Definitions

int typeptr = TYPE_INIT;
tnode types [MAXTYPES];

Vartable Definitions

varptr varhead = NULL;

Deftable Definitions

defptr defhead = NULL;

And List Definitions

and_ptr and_head = NULL;
and_flag = FALSE;

Makeform

fnode
makeform ()
{
    return ((fnode) alloca (1, sizeof (fnode)));
}

Putform

void
putform (type)
{
    PHITYPE type;
    extern fnode *head;
    fnode *ptr = makeform (),
    *tracer;

    // Placement code goes here...
}
ptr->type = type;
if (head != NULL) {
  tracer = head;
  while (tracer->link != NULL)
    tracer = tracer->link;
  tracer->link = ptr;
  ptr->link = NULL;
} else {
  head = ptr;
  ptr->link = NULL;
}

/*******************************************************************************
** Makevar ***************************************************************
*******************************************************************************/
varptr
makevar ()
/* Make node for vars linked list */
return (struct varnode*)
calloc (1, sizeof (struct varnode));

/*******************************************************************************
** Putvar ******************************************************************
*******************************************************************************/
void
putvar (type, treenode)
/* Put variable in variable */
TYPE type;
node treenode;
extern int form;
varptr ptr = makevar ();
ptr->nptr = treenode;
ptr->type = type;
ptr->form = form;
ptr->link = varhead;
varhead = ptr;
ptr = NULL;
free (ptr);

/*******************************************************************************
** Findvar ******************************************************************
*******************************************************************************/
varptr
findvar (varname)
/* Find var in variable */
long varname;
varptr ptr = varhead;
while (ptr != NULL) {
  if (ptr->nptr->index == varname)
    return (ptr);
  ptr = ptr->link;
}
return (NULL);

/****-----------------------------------***/
****-----------------------------------*****/
/]*************** Getvtype ***************/

PHITYPE
getvtype (ptr) /* Get type of var in var stack */
    { varptr ptr;
      return (ptr->type);
    }

/]*************** Putdef ***************

void
putdef (type, treeptr) /* Put var in definitions table */
    { PHITYPE type;
      nodal treeptr;
      extern int form;
      defptr ptr = (struct defnode*)malloc(sizeof (struct defnode));
      ptr->nptr = treeptr; /* Fill entry */
      ptr->type = type;
      ptr->link = defhead;
      defhead = ptr;
      ptr = NULL;
      free (ptr);
    }

/]*************** Finddef ***************

defptr
finddef (varname) /* Find var in deftable */
    { long varname;
      defptr ptr = defhead;
      while (ptr != NULL) {
        if (ptr->nptr->index == varname) /* Break if variable found */
          return (ptr);
        ptr = ptr->link;
      }
      return (NULL); /* No tally on variable */
    }

/]*************** getfptr ***************

node
*getfptr (ptr) /* Return fptr from def table */
    { defptr ptr;
      return (ptr->fptr);
    }

/]*************** Getdtype ***************

PHITYPE
getdtype (ptr) /* Get type of var in def table */
    { defptr ptr;
      return (ptr->type);
    }

/]*************** Add_and ***************

void
add_and (ptr) /* Add and node to and list */
    { nodal ptr;
    /* Ptr to node containing var */
    }
extern and_ptr and_head, and_alloc();
extern int buff_ptr;
ad_ptr a_ptr = and_alloc(); /* Holder for and_ptr */
a_ptr->buff_ptr = buff_ptr; /* Set ptr to current buffer ptr */
a_ptr->ptr = ptr; /* Set ptr to node with var set */
a_ptr->link = and_head; /* Link node to list */
and_head = a_ptr; /* Dispose of a_ptr */

/********************************************************** And_Alloc **********************************************************/
ad_ptr and_alloc() /* Create a node for and list */
return ((struct and_struc*)alloc(1, sizeof(struct and_struc)));

/********************************************************** Del_and **********************************************************/
void del_and (ptr) /* Delete entry into the and_list */
and_ptr ptr;
extern and_ptr and_head;
ad_ptr search = and_head;
if (ptr = and_head) /* Case if pointer not equal to */
while (search->link != ptr) /* first entry in list */
search = search->link; /* Place ptr on entry above */
search->link = ptr->link; /* Get entry */
else and_head = ptr->link; /* Case ptr to list entry in list */
ptr->link = NULL; /* Dispose of unneeded node */
free (ptr);

/********************************************************** Terror **********************************************************/
void terror (err_num, line_num) /* Sem check error handling */
/* routine */
int err_num, line_num;
extern ErrorHandler ();
err_found = TRUE; /* Set err_found to true */
ErrorHandler (line_num, err_num, SEM_NOT); /* stop code gen */
/* generic error handling ptr */

/********************************************************** Putaddr **********************************************************/
void put_addr (ptr, type) /* Inserts virtual address */
/* variable function return */
/* And increments burn list */
/* Assumes current hack list */
node_ptr; /* Pointer to target node */
**PHITYPE** type;

ptr->addr = curr_addr;

ptr->scope = curr_scope;

curr_addr = curr_addr + (types [type].bytes);

if (curr_addr > MAXADDR)

    error (ERR_mm, ptr->ln);

/**************************** Name *****************************/

char *name ()

char *string = malloc (7),

    *string1 = malloc (7);

static long seed = 10000;

    *string = 'a';

    *(string - 1) = ENDSTRING;

    strcat (string, string1);

    --seed;

    return (string);

/**************************** Formal *****************************/

FLAG formal (ptr) /* Returns true if the varnode */

    varptr ptr;

    if (ptr->form) return (TRUE);

    else return (FALSE);
APPENDIX K
ROCK COMPILER — CODE GENERATION MODULE

PUBLIC DOMAIN SOFTWARE

Name: Code Generation Module
File: Code_GEN.c
Authors: Maj E.J. COLE / Capt J.E. CONNELL
Started: 02/06/87
 Archived: 04/10/87
 Modified: 04/13/87 Code output to vdisk ET

This file contains the following modules for the PHI compiler:

C_Store_Code  C_Startup  C_Init_Insert
C_Ending     C_Printcode  C_Init
AcCode       C_Jmp       C_Start_Proc
C_I_Const    C_I_Form    C_End_Proc
C_I_Op       C_Call_Proc

Algorithm:
This module contains the procedures necessary for code generation:
C_Startup initializes the run_time file, & the semantic checker & call the procedures as necessary. Note that "c_store_code" is a generic generator which will spew any string given as an arg to the output file.

Modified: 04/13/87 Code output to vdisk, drive "a:" ET

Externals

include <semconv.h>
include string.h
include start.h

extern char code[56];
extern long code_add;

Globals

int code_buffer;
char * code_ptr = NULL;

C_Store_Code

void
store code string:

ptr = NULL

137
if (!err_found) { /* Compute only if no error found */
    while (*string != NULL) { /* Copy string char by char */
        *code_buffer + buff_ptr) = *(string + ptr);
        ++ptr;
        ++buff_ptr;
    }
}

/******************************* C_Jmp *****************************/
void
C_Jmp (name) char *name;
    /* Gen code to insert 'jmp command' */
    c_store_code("jmp ");
    c_store_code(name);
    c_store_code("\n");

/******************************* C_Start_Proc *******************************/
void
C_Start_Proc (name) char *name;
    /* Output name for start of asm */
    /* language procedure */
    c_store_code(name);
    c_store_code("\n");

/******************************* C_End_Proc *******************************/
void
C_End_Proc (name) char *name;
    /* Output name for ending an */
    /* assembly language procedure */
    c_store_code("call del_scope\n");
    c_store_code("ret\n");
    c_store_code(name);
    c_store_code("\n");

/**************************** C_Call_Proc *****************************/
void
C_Call_Proc (name) char *name;
    /* Output call for an assembly */
    /* language procedure */
    c_store_code("call ");
    c_store_code(name);
    c_store_code("\n");

/**************************** C_1_Form *****************************/
...
/*************** C_I_Const ***************

void c_i_const (name) /* Output code for assigning an integer constant */
{
    char *name;
    c_store_code ("mov ax,");
    c_store_code (name);
    c_store_code ("\n");
    c_store_code ("call iputvalue\n");
}

/*************** C_I_Op ***************

void c_i_op (op) /* Output code for int arith ops */
{
    optype op;
    extern void terror ();
    switch (op) {
        case (ADD) : c_call_proc ("iadd");
                     break;
        case (SUB) : c_call_proc ("isub");
                     break;
        case (DIVIDE) : c_call_proc ("idivn");
                         break;
        case (MULT) : c_call_proc ("imult");
                     break;
        default : return;
    }
}

/******* Startup ***********/

void c_startup () /* Open and initialize files */
{
    code_buffer = getmen (SIZEBUFFER);
    c_store_code ("extrn initial : near\n");
    c_store_code ("extrn ladd : near\n");
    c_store_code ("extrn isub : near\n");
    c_store_code ("extrn imult : near\n");
    c_store_code ("extrn idivn : near\n");
    c_store_code ("extrn lequ : near\n");
    c_store_code ("extrn lneq : near\n");
    c_store_code ("extrn lgt : near\n");
    c_store_code ("extrn llt : near\n");
    c_store_code ("extrn land : near\n");
    c_store_code ("extrn lor : near\n");
    c_store_code ("extrn igteq : near\n");
    c_store_code ("extrn iputvalue : near\n");
    c_store_code ("extrn iteq : near\n");
    c_store_code ("extrn igteq : near\n");
    c_store_code ("extrn iadd : near\n");
    c_store_code ("extrn idivn : near\n");
    c_store_code ("extrn iadd : near\n");
    c_store_code ("extrn idivn : near\n");
    c_store_code ("extrn negation : near\n");
    c_store_code ("extrn i_formal : near\n");
    c_store_code ("extrn i_mov : near\n");
    c_store_code ("extrn ppush : near\n");
    c_store_code ("extrn ppop : near\n");
    c_store_code ("extrn add_scope : near\n");
    c_store_code ("extrn del_scope : near\n");
    c_store_code ("org 0100h\n");
    c_store_code ("cseg\n");
    c_store_code ("call initial\n");
}
/* *********************** C_Print_Code ************************/
void
c_print_code ()
    /* Output code buffer to */
    /* secondary storage */
(extern char prefix []);
int code;
char holder[30];
strcpy (holder, "d:");
strcat (holder, prefix);
strcat (prefix, holder);
strcat (holder, "a.86");

code = open(FILENAME,O_TRUNC | O_WRONLY,NULL);
    /* Open file for writing and */
    /* overwriting only */
write (code, code_buffer, buff_ptr);
    /* Write the buffer */
close (code);
    /* Close the output file */

/* *********************** C_Ending ************************/
void
c_ending ()
    /* Ending for output code */
{
    if (!err_found) {
        c_store_code ("call print_top\n");
            /* Print address pointed to by */
        c_store_code ("call finis\n");
            /* Routine to make clean ending */
        
            "(code_buffer + (buff_ptr +)) = CNTRLZ;
            /* If no error, put asm language */
            /* delimiter to file */
        c_print_code ();
            /* Output code to a file */
    }

/* *********************** c_ztor ************************/
void
c_ztor ()
    /* Gen code for conv int to real */
();
    /* Empty now, but watch our smoke */

/* *********************** Acode ************************/
void
acode (ptr, type)
    /* NOTE : USES EMPTY STATEMENTS */
    /* FOR REAL OPERATIONS */
{extern void terror ()};
int name;

name = ptr->name;

switch (name) {
    case (ADD) : if (type == REAL);
             else c_i_op (ADD);
                    break;
    case (SUB) : if (type == REAL);
             else c_i_op (SUB);
                    break;
    case (MULT) : if (type == REAL);
             else c_i_op (MULT);
                    break;
    case (DIV) : if (type == REAL);
              break;
    case (RDIV) : if (type == REAL);
              break;
    }
case (DIV): c_i_op (DIVIDE);
    break; /* Integer Division */
APPENDIX L
ROCK COMPILER — USER INTERFACE

/* Name : User Interface
* File : User.C
* Authors : Maj E.J. COLE / Capt J.E. CONNELL
* Started : 04/01/87
* Archived : 04/10/87
* Modified :

This file contains the following modules for the PHI compiler

User_err Getname Prog_name
Print_header P_Close User

Algorithm :
This module contains the procedures necessary for the user interface.
Prog_Name gets the user's choice of program by calling Get_Name.
Print header is called to print the initial screen display on console, & the User procedure is the overall coordinator of the interface.
User_Err and P_Close are both independent procedures. User_Err handles output in the event that an error or errors have been found.*
P_close is called by "Rock_Main" to ensure the input file has been closed.

Modified :

#include <user.h>
#include <dos.h>
#include <stdio.h>
extern void clrscr (), mov_cursor (), clr_window ();

char s_name [BUFFLENGTH], /* Name of Source file */
prefix [BUFFLENGTH]; /* Prefix of source file */
FILE *infil; /* File handle of source file */

void
User_Err (); /* Screen interface for error */

142
extern int num_errors;
FILE *errors;
int numblocks,
count = 0;
char *buffer = malloc (BSIZE),
input;

errors = fopen (ERRORFILE,"a");
fornt (errors, 
  "Number of errors = %d\n",num_errors);
putc ('$', errors);
fclose (errors);

cirsc ()
errors = fopen (ERRORFILE, "r");
umblocks = fread(buffer,BLOCKSIZE,20,errors); /* Read error msg from error files*/ 
BLOCKSIZE will allow whole 
file to be read at once */
while (*buffer + count) != '$' { 
  putchar (*buffer + count);
  ++count;
}
printf ("\n \n \n");
printf ("%s", PAUSE);
input = getch ();
fclose (errors);
cirsc ();
if (input == ESCAPE) exit (1);

/*************************** Getname ***************************/

void
getname ()

(int ch,
count = 0;

do {
  if ((ch = getch ()) == BACKSPACE) {
    if (count) { --count;
      putchar (ch);
      putchar ('\'');
      putchar (ch);
    }
  }
  else if (ch == ESCAPE) {
    cirsc ();
    exit (1);
  }
  else if (ch < 127)
    putchar (ch);
  name [count] = ch;
  ++count;
}
while (count <= BUFSIZE) &&
ch != EOLN);
**PROG_NAME**

```c
void prog_name()
{
    do
    {
        clr_window (9,1,21,79);
        mov_cursor (10,2);
        printf (GETPROGRAM);
        getname ();
        infile = fopen (u_name, "r");
        if (!infile) {
            mov_cursor (20,33);
            printf (FILE1_ERROR);
            printf (FILE2_ERROR);
            if (getch () == ESCAPE) {
                clrscr ();
                exit (1);
            }
        }
        while (!infile);
        mov_cursor (13,28);
        printf (WAIT);
}
```

**Print_header**

```c
void print_header()
{
    clrscr ();
    mov_cursor (1,33);
    printf (HEADER1);
    mov_cursor (2,24);
    printf (HEADER2);
}
```

**P_Close**

```c
void p_close()
{
    fclose (infile);
}
```

**User**

```c
void user()
{
    int count = 0;
    printf (HEADER1);
    printf (HEADER2);
    printf (WAIT);

    while ((u_name [count] == ',')
        && (u_name [count] == NUL)) {
        prefix [count] = u_name [count];
        count++;
    }
    prefix count = 0;
}
```
APPENDIX M
ROCK COMPILER — RUNTIME UTILITIES

;**********************************************************************;
; Name : Phi Runtime Utilities
; File : U.a86
; Authors : Maj E.J. COLE / Capt J.E. CONNELL
; Started : 01/26/87
; Archived : 16 Feb 87
; Modified : 16 Apr 87 Stack/VarSpace Crash error check EC
;**********************************************************************;

/ALGORITHMS

1. Input/Output: The first section of the program contains input and output.

2. Virtual Space: A virtual space is set up in the extra segment to hold both the
stack. The middle of this space is denoted by the symbol "vars", and variables
offset (± 32700) from vars. In this implementation, the program stack grows from
vars grow from the bottom. The virtual space is assumed to be made up of words
two bytes), so only
even numbers may be used to access it.

3. Stack: The stack pointer is the si register, which is initialized to 32700.
grows, the si register is reduced by two. Push and pop will push and pop two
registers. "Push_one" and "Pop_one" will push and pop single words to and from

4. Addressing Program Variables: Each program variable is assigned a two-tuple A
scope and 0 is the offset from the base address of variables in that scope.
turn the address of a variable given A.

5. Scoping: Initially the scope is set to 0; the global scope. The variable
space containing the outer scope, and the variable "S_Nest" contains the current
new scope is created, "S_Nest" is increased by one, and the three-tuple S =
(1 = Static Link, pointing nesting level of the outer scope, N is the nesting
is the base address of display of variables for this scope.
When a scope is deleted, the top of the stack is saved, the top instantiation of 3
and S_Link and S_Nest are recalculated.

6 Inserting/Extracting Program Variables: "I_Assign" will insert an integer or
scope contained in S_Nest when it is requested. "InputValue" will insert the
responding tuple A on the stack. "IetValue" will pop the tuple A off the top of
the value of the integer pointed to by A.

;**********************************************************************;

;**********************************************************************

; Modified : 22 Feb 87 Add/del_scope changed to save TOS. EC
; 16 Apr 87 Added check for stack/varspace crash, includes
message to observer

145
Public Procedures

Public i mov
Public i_formal
Public i_getvalue
Public finis
Public inputvalue
Public find_addx
Public add_scope
Public del_scope
Public initial
Public finis
Public ppush
Public ppop
Public lassign
Public lor
Public land
Public lequ
Public ineq
Public ltd
Public lgt
Public ilteq
Public igteq
Public negation
Public ladd
Public isub
Public imult
Public idivn
Public print_top

I/O Procedures

print_char:
Print a char to the screen
assumes letter to be printed is in dl register

Print Num
Prints, as a number, the value found in the bx register
```assembly
print_num:    push ax
            push bx
            push cx
            push dx
            mov cx, 10000
            cmp cx, 0
            jge small
            mov dx, '-'
            call print_char
            neg dx

small:     cmp bx, 10
           jl final

div_loop:  mov ax, bx
            xor dx, dx
            div cx
            cmp ax, 0
            jte p_loop
            mov ax, cx
            mov cx, 10
            xor dx, dx
            div cx
            mov cx, ax
            jmp div_loop

p_loop:    mov ax, bx
            xor dx, dx
            div cx
            mov bx, dx
            add ax, 48
            mov dx, ax
            call print_char
            xor dx, dx
            mov ax, cx
            mov cx, 10
            div cx
            mov cx, ax
            cmp ax, 1
            jne p_loop

final:     add bx, 48
            mov dx, bx
            call print_char
            call echo
            pop dx
            pop cx
            pop bx
            ret

;**************** Print_top **************************
;Prints the space pointed to by the top tuple of the program stack
;
print_top:  mov di, si
            add di, 1
            mov bx, vars di
            add di, 2
            mov cx, vars di
            ;mov offset to cx
```

147
call find_addr
mov di, cx
mov bx, vars [di]
call print_num
call end
ret

;*********************************************** print_s ***********************************************
;assumes address of is in the dx register
;assumes string ends with a "$" sign
;
print_s:
push ax ;save register
mov ah, 9
int 21h
pop ax
ret

;************************************************************************ Stack Procedures *
;************************************************************************

;*********************************************** Ppush ***********************************************
;Pushes values from cx (offset) and di (nesting level)
;
push:  mov vars [si], cx
sub si, 2
mov vars [si], di
sub si, 2
cmp si, curr_addr
jg p_return
mov dx, offset crash
call print_s
call finish

p_return: ret

;*********************************************** Push_one **********************************************
;Push a single integer from cx register to the program stack
;
push_one:  mov vars [si], cx
sub si, 2
ret

;*********************************************** PPop **********************************************
;Pop values from the program stack to di (nesting level) and cx (offset)
;
pop: add si, 2
mov di, vars [si]
add si, 2
mov cx, vars [si]
ret

;*********************************************** Pop_one **********************************************
;Pop a single integer from the stack to the cx register
;
pop_one: add si, 2
mov cx, vars [si]
ret
Varpace Management Procedures

IAssign

Assign an integer value to a variable space in current scope
Assumes value is in ax; offset is set to current max offset

; Iassign: mov dl, s_link
; get static link
sub dl, 2 ; decrement it to pt to case address
mov dl, var's_di
add dl, max_offset
mov vars_di, ax
add max_offset, ax
add curr_addr, 2
ret

IGetValue

Pop the stack and move the integer value pointed to into the ax register

; Igetvalue: call ppop;
mov dx, dl
call find_addr
mov dl, cx
mov ax, var's_di
ret

IPutvalue

Takes an integer from AX register, puts its value into varspace, then puts its address on the top of the stack

; Iputvalue: mov dx, s_nest
mov cx, max_offset
call find_addr
mov dl, cx
mov vars_di, ax
mov dl, s_nest
mov cx, max_offset
call ppush
add max_offset, 2
add curr_addr, 2
ret

Scoping Procedures

Find Address

Returns address of variable at nesting level dx, offset cx to cx reg

; Find_addr: mov dl, s_link
; ;get addr of current static level
find_loop: cmp es:vars_di, cx
je 'exit
add di,2
mov di, es:vars[di]
jmp find_loop

f_out: sub di,2
add cx, es:vars[di]
ret

;************************** Add_Scope ******************************
;Start new scope by adding static link, starting address, & nesting level
;
add_scope: mov cx, s_link
inc s_nest
mov di, s_nest
call ppush
mov cx, curr_addr
call ppush
mov max_offset, 0
mov s_link, si
add s_link, 6
ret

;************************ Del_Scope ******************************
;Deletes a scope
;
del_scope: call ppop;
mov cx, di
call find_addr
push cx
dec s_nest
mov si, s_link
sub si, 4
mov cx, es:vars [si]
mov max_offset, cx
mov bx,2
mov cx, es:vars [si-bx]
mov curr_addr, cx
add si, 6
mov cx, es:vars [si]
mov s_link, cx
pop di
mov ax, es:vars [di]
call inputvalue
ret

;*******************************************************************
;*
;*
;*******************************************************************

;*******************************************************************
;Begin/End Procedures
;*******************************************************************

;*******************************************************************
;Initial  ******************************************************
;initialize the stack and variables
;must initialize cx to base of stack heap before calling this
;
initial:  mov si, SPACE_JCP
          mov di,0
          mov cx, 0
call ppush
ret
;********************** finis **********************

; finis:
mov ax,04c00h
int 21h
ret

;********************** Booleans **********************

;********************** Negation **********************
;Negates a boolean value
negation:
call igetvalue
cmp ax,1
jne zero
mov ax,0
jmp p
zero:
mov ax,1
p:
call iputvalue
ret

;********************** Lor **********************
;Takes logical or of two boolean values and stacks address of answer
lor:
call igetvalue
mov bx,ax
call igetvalue
or ax,ax
call iputvalue
ret

;********************** Land **********************
;Takes logical and of two boolean values and stacks address of answer
and:
call igetvalue
mov bx,ax
call igetvalue
and ax,ax
ret

;********************** Iequ **********************
;Takes logical equal of two integers and stacks address of answer
i eq:
call igetvalue
mov bx,ax
call igetvalue
eq ax, bx
mov ax,0
jmp p
p:
call iputvalue
ret

mov ax,0
;*********************************************************************** Ineq ************************************************************************
;Takes logical not equal of two integers and stacks address of answer

;eq:
  call .getvalue
  mov cx, ax
  call .getvalue
  cmp ax, cx
  jne eq
  mov ax, FALSE
  ret

;eq:
  mov ax, TRUE
  ret

;*********************************************************************** Ilt ************************************************************************
;Takes logical less than of two integers and stacks address of answer
;Returns true if first value is less than the second value

;lt:
  call .getvalue
  mov ex, ax
  call .getvalue
  cmp ax, ex
  jg less
  mov ax, TRUE
  ret

;lt:
  mov ax, FALSE
  ret

;*********************************************************************** Igt ************************************************************************
;Takes logical greater than of two integers and stacks address of answer
;Returns true if first value is greater than the second value

;gt:
  call .getvalue
  mov ex, ax
  call .getvalue
  cmp ax, ex
  jle greater than
  mov ax, TRUE
  ret

;gt:
  mov ax, FALSE
  ret

;*********************************************************************** Ilteq ************************************************************************
;Takes logical less than or equal of two integers and stacks address of answer
;Returns true if first value is less than or equal to the second value

;lteq:
  call .getvalue
  mov ex, ax
  call .getvalue
  cmp ax, ex
  jg ilteq
  mov ax, FALSE
  ret

;lteq:
  mov ax, TRUE
  ret

152
mov ax, TRUE
con2: call iputvalue
ret

ltdq: mov ax, FALSE
jmp con2
ret

;*************************** Igteq ****************************
;Takes logical 2 of two integers and stacks address of answer
;Returns true if first value is greater than or equal to the second value

Igteq: call igetvalue
mov bx, ax
call igetvalue
cmp ax, bx
jl ltdq
mov ax, TRUE
con3: call iputvalue
ret

Igteq: mov ax, FALSE
jmp con3
ret

;******************************************************************************
; Integer Operations
;******************************************************************************

;******************************************************************************
; Iadd **************************************************************************
; Adds two integer values
; Assumes offset of second value is in SI register
; Offset of first value is at the top of the stack

Iadd: call igetvalue
mov bx, ax
call igetvalue
add ax, bx
jz err
call iputvalue
ret

Iadd: mov ax, offset add_err
        call print_s
        call echo
        call find
        ret

;******************************************************************************
; ISub **************************************************************************
; Subs two integer values
; Assumes offset of second value is in SI register
; Offset of first value is at the top of the stack

ISub: call igetvalue
mov bx, ax
call igetvalue
sub bx, ax
jz err
        call iputvalue
        ret

ISub: mov ax, offset add_err
        call print_s
        call echo
        call find
        ret
jo errs
; if overflow, run time error

call iputvalue
ret

errs:  mov dx, offset sub_err
; Print error message on overflow
call print_s
call eoln
call finishes
ret

;****************************** IMult ******************************
; Multiplies two integer values
; Assumes offset of second value is in SI register
; Offset of first value is at the top of the stack

imult:
call igetvalue
mov bx, ax ; First value to cx register
call igetvalue
imul bx ; Perform mul, result in AX
jc err1 ; if carry set, run time error

call iputvalue
ret

err1:  mov dx, offset mul_err
; Put integer into varspace
call print_s
call eoln
call finishes
ret

;****************************** IDivn ******************************
; Divides two integer values, result in varspace, address of result stacked
; Offset of first value is at the top of the stack

idivn:
push cx ; Save Registers
push dx
call igetvalue
mov bx, ax
; mov divisor to bx
call igetvalue
xor dx, dx ; Set dx to 0
mov cl, 1
mov ch, 1
cmp bx, 0
; cl and ch are negative flags
; bx is positive, no action needed
je test2
je errd
neg cl
cmp ax, 0
; bx is 0, ERROR
neg bx

; bx is negative, cl flag negated
neg bx

; bx is made positive

; test dividers
; dividend >= 0, no action
; dividend < 0, ax flag negated
; ax is made positive

test2:
sub ax, bx
; Loop and count subtractions
cmp ax, 0
jg done
jmp divloop

inc dx
jmp divloop

done:
ret
done: mov al, cl
mul ch
cmp al, 0
jge dend
neg dx
dend: mov ax, dx
pop dx
pop cx
call inputvalue
; Put integer into varspace
;
errd: mov dx, offset div_err
call print s
call eoln
call finish
ret

; Function Calling Procedures

;**************************************************
;
; i_mov
; Moves integer or boolean actuals with addresses at the top of stack to
; the lowest addresses within a scope
; Assumes bx has number of actuals needed to be moved

i_mov: pop ret_addr
; Save i_mov's return address
call add_scope
; mov addresses to ox and dx: regs
setq
call find_addr
mov dl, cx
mov ax, es:vars [dl]
call lassign
dec bx
cmp bx, 0
jne str
push ret_addr
; Restore i_mov's return address
ret

; I_formal
; Puts a formal to the top of the stack
; Assumes offset of formal in cx register

i_formal: mov dl, 0
; Get context level.
call ppush
; Push offset and rest into stack
ret

; seq
;***************************** Constants ****************************

TRUE EQU 1  
FALSE EQU 0  
SPACE_TOP EQU 32700 ;Top of memory space

;***************************** Integer Variables ****************************

max_offset dw 0 ;Maximum current offset w/in scope  
curr_addr dw -32700 ;Current maximum address  
s_link dw SPACE_TOP ;Current address of static link  
s_nest dw 0 ;Current static nesting level  
ret_addr dw 0

;***************************** Error Messages ****************************

div_err db 'DIVISION BY ZERO, FOOL!'  
db 'S'

mul_err db 'MULTIPLICATION OVERFLOW, IDIOT!'  
db 'S'

add_err db 'ADDITION OVERFLOW, DUMMIE!'  
db 'S'

sub_err db 'SUBTRACTION OVERFLOW, NITWIT!'  
db 'S'

crash db 'STACK/VARIABLE SPACE CRASH'  
db 'S'

;***************************** Error Messages ****************************

eq seq vars dw 0
end
APPENDIX N – TEST SUITE

SIMPLE TESTS OF FUNCTIONS AND VARIABLES

let c : $Z -> $Z;
c (20) where c (n) == if 1 = 2 then 3 * n
else 3 + n endif

-- Simple "Hello I'm Alive Test"

let c : $Z -> $Z;
c (1 * 2) where c (n) == n * 3

-- Test for expression in functions's formals

let c : $Z -> $Z;
c (k + 2) where k == 2 and
c (n) == if n = 1 then n * 3 else n + 4 endif

-- Test for expression in function's formals

TESTS FOR RECURSION

let c : $Z -> $Z;
c (k * 2) where k == 2 and c (n) == n * 3

-- Test for expression in function's formals

let c : $Z -> $Z;
c (0) where c (n) == if n = 0 then 1 else c (n - 1) * n endif

-- Test for recursion in functions

let c : $Z -> $Z;
c (5) where c (n) == if n = 0 then 1 else c (n - 1) * n endif

-- Test for recursion in functions

let c : $Z -> $Z;

157
c (3) where c (n) == if n = 0 then 1 else n * c (n - 1) endif

-- Test for recursion in functions

let c : $Z -> $Z;

let c : $Z -> $Z;
c (7) where c (n) == if n = 0 then 1 else n * c (n - 1) endif

-- Test for recursion in functions

TESTS OF COMPLEX FUNCTIONS, INCLUDING BOOLEANS AS ARGUMENTS AND RESULTS

let c : $Z -> $B;

c (1) where

\[ c (n) == n = 6 \]

-- Test for booleans in function

let c : $Z * $Z * $Z -> $Z;

c(2 - 1,3,4) where c(n,m,x) == n * m * x

-- Test for multiple arguments

let c : $Z -> $B;

let d : $Z -> $Z;

c (1) where

\[ c (n) == 1 = d(1) \] where

\[ d(k) == k \]

-- Test for chaining in functions

let c : $Z -> $Z;

tet d : $Z -> $Z;

let e : $Z -> $B;

c (3) where

\[ c (n) == 1 + d(n) \] where

\[ d(k) == if e(1) then k else k + 1 endif \]

\[ e (k) == k = 3 \]

-- Test for nesting in functions
let \( c : \mathbb{Z} \rightarrow \mathbb{Z} \);
let \( d : \mathbb{Z} \rightarrow \mathbb{Z} \);
let \( e : \mathbb{Z} \rightarrow \mathbb{B} \);

\[
c (3) \times 10 \text{ where } c \left( n \right) = 1 + d \left( n \right) \text{ where } d \left( k \right) = \text{if } e \left( 1 \right) \text{ then } k \text{ else } k + 1 \text{ endif } \text{ where } e \left( k \right) = k = 3
\]

-- Test for nesting in functions, result multiplied by constant

let \( c : \mathbb{Z} \rightarrow \mathbb{Z} \);
let \( d : \mathbb{Z} \rightarrow \mathbb{Z} \);
let \( e : \mathbb{Z} \rightarrow \mathbb{B} \);

\[
c (3) \times c \left( 4 \right) \text{ where } c \left( n \right) = 1 + d \left( n \right) \text{ where } d \left( k \right) = \text{if } e \left( 2 = 3 \land 4 = 5 \right) \text{ then } k \text{ else } k + 1 \text{ endif } \text{ where } e \left( k \right) = k
\]

-- Test for two functions, same definition
-- Also, test for extraneous variable defined at end of program

let \( c : \mathbb{Z} \rightarrow \mathbb{Z} \);
let \( d : \mathbb{Z} \rightarrow \mathbb{Z} \);
let \( e : \mathbb{B} \rightarrow \mathbb{B} \);

let \( b = 10 \)

\[
c (3) \times b \text{ where } b = 10 \text{ and } c \left( n \right) = n \times d \left( n \right) \text{ where } d \left( n \right) = 3
\]

-- Test for nesting in functions

TESTS FOR "AND" AND "WHERE" NESTING AND COMBINATIONS

let \( c : \mathbb{Z} \rightarrow \mathbb{Z} \);
let \( d : \mathbb{Z} \rightarrow \mathbb{Z} \);

\[
c (3) \times b \text{ where } b = 10 \text{ and } c \left( n \right) = n \times d \left( n \right) \text{ where } d \left( n \right) = 3
\]

-- Test for nesting in functions
let c : $Z -> $Z;
let d : $Z -> $Z;

\[ c(3) \times b \text{ where } b = 10 \text{ and } c(n) = n \times d(n) \text{ where } d(n) = 3 \times e \text{ where } e = 10 \]

-- Test for nesting in functions

let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;

\[ c(3) + b \text{ where } b = 10 \text{ and } c(n) = d(1) + \text{if } n = e(1) \text{ then } 2 \text{ else } 10 \text{ endif} \]
\[ \text{where } e(k) = 1 \text{ and } d(g) = g + 5 \]

-- Test for nested wheres and ands

let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $B;

\[ c(3) \text{ where } c(n) = 1 + d(n) \text{ where } d(k) = \text{if } e(1) \text{ then } k \text{ else } k + 1 \text{ endif} \]
\[ \text{where } e(b) = b = 3 \]

-- Test for nesting in functions

let c : $Z -> $Z;
let d : $Z;

\[ c(5) \text{ where } c(n) = d \text{ and } d = 10 \times 5 \]

-- Test for single and statement

-- Test for datadef declaration

let c : $Z;
let d : $Z;
let e : $Z;

\[ c \text{ where } c = (d + 10 + e \text{ where } e = 10) \]
and d == 10
-- Test for Multiple ands

let c : $Z;
let d : $Z;
let e : $Z;

c where c == d + 10 + e
    and d == 10
    and e == 10
-- Test for Multiple ands

let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
c(5) where c(n) == d(n) + 12
    and d(s) == 10 + s
-- Test for Multiple ands using functions

let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
c(5) where c(n) == d(n) + 12
    and d(s) == 10 + e (s)
    and e(k) == 20 + k + t where t == 100
-- Test for Multiple ands, nested wheres

let c : $Z;
let d : $Z;
let e : $Z;
c where c == d + 10 + e where
    e == 10 and d == 10
-- Test for Multiple ands

let c : $Z -> $B;
let d : $Z -> $B;
let k : $Z -> $Z;
c(1) ∧ d(2) where
c(n) = n = 3 and
d(n) = (1 = k (n - 1) where
k (1) = l + 10)

-- Test for proper use of "and" and implementation of
-- Parens

let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
c(5) where c(n) = d(n) + 12 where k = 100
and d(s) = 10 + e (s) and e(k) = 20 + k

-- Test for Multiple ands, multiple wheres and formal/variable collisions

let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
c(5) where c(n) = d(n) + 12 where k = 100
and d(s) = 10 + e (s) where t = 100
and e(k) = 20 + k + t

-- Test for Multiple ands, multiple wheres and formal/variable collisions

let c : $Z -> $Z;
let d : $Z -> $Z;
let e : $Z -> $Z;
c(5) where c(n) = d(n) + 12 where t = 100
and d(s) = 10 + e (s) where t = 120
and e(k) = 20 + k + t

-- Test for Multiple ands, multiple wheres and formal/variable collisions
-- Also test to see if the proper "t" (120) was picked up

let c : $Z * $Z -> $Z;
let d : $Z * $Z -> $Z;
let e : $Z * $Z -> $Z;
\[ c(5,1) \text{ where } c(n,m) = d(n,m) + 12 \text{ where } t = 100 \]
and \[ d(s,z) = 10 + e(s,z) \text{ where } t = 120 \]
and \[ e(k,l) = 20 + k + t + l \]

-- Test for Multiple ands, multiple wheres and formal/variable collisions

-- Test specifically for functions with multiple arguments

let \( c : \mathbb{Z} \to \mathbb{Z} \); let \( d : \mathbb{Z} \to \mathbb{Z} \); let \( e : \mathbb{Z} \to \mathbb{Z} \);

\[ c(5) \text{ where } c(n) = d(n) \text{ where } t = 100 \]
and \[ d(s) = e(s) \text{ where } k = 2 \]
and \[ e(k) = 20 + t \]

-- Test for Multiple ands, multiple wheres and formal/variable collisions

let \( c : \mathbb{Z} \to \mathbb{Z} \); let \( d : \mathbb{Z} \to \mathbb{Z} \); let \( e : \mathbb{Z} \to \mathbb{Z} \);

\[ c(10) \text{ where } c(n) = d(n) \text{ where } t = 100 \]
and \[ d(s) = e(s) \text{ where } k = 10 \]
and \[ e(r) = 20 + r + k \]

-- Test for Multiple ands, multiple wheres and formal/variable collisions

let \( c : \mathbb{Z} \to \mathbb{Z} \); let \( d : \mathbb{Z} \to \mathbb{Z} \); let \( e : \mathbb{Z} \to \mathbb{Z} \);

\[ c(10) \text{ where } c(n) = d(n) + t \text{ where } t = (r \times 100 \text{ where } r = 2) \]
and \[ d(s) = e(s) \text{ where } k = 10 \]
and \[ e(r) = 20 + r + k \]

-- Test for Multiple ands, multiple wheres and formal/variable collisions

let \( c : \mathbb{Z} \to \mathbb{Z} \); let \( d : \mathbb{Z} \to \mathbb{Z} \); let \( e : \mathbb{Z} \to \mathbb{Z} \); let \( f : \mathbb{N} \to \mathbb{Z} \);

\[ c(10) \text{ where } c(n) = d(n) + t \text{ where } t = (r \times 100 \text{ where } r = 2) \]
and \[ d(s) = e(s) \text{ where } k = 10 \]
and \[ e(r) = 20 + r + f(r) \]
and \( f(r) = r \)

-- Test for Multiple ands, multiple wheres and formal/variable collisions

let \( c: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( d: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( e: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( f: \mathbb{N} \rightarrow \mathbb{Z}; \)

\[
c(10) \text{ where } c(n) = d(n) + t \text{ where } t = (r * 100 \text{ where } r = 2) \\
\text{and } d(s) = e(s) \text{ where } k = 10 \\
\text{and } e(r) = 20 + r + f(r) \\
\text{and } f(r) = k
\]

-- Test for Multiple ands, multiple wheres and formal/variable collisions

let \( c: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( d: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( e: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( f: \mathbb{N} \rightarrow \mathbb{Z}; \)

\[
c(10) \text{ where } c(n) = d(n) + t \text{ where } t = (r * 100 \text{ where } r = 2) \\
\text{and } d(s) = e(s) \text{ where } k = 10 \\
\text{and } e(r) = 20 + r + f(r) \\
\text{and } f(r) = \text{if } r = 0 \text{ then } 100 \text{ else } f(r - 1) \text{ endif}
\]

-- Test for Multiple ands, multiple wheres and formal/variable collisions

-- Test for if-then-else collisions with multiple ands, wheres

let \( c: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( d: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( e: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( f: \mathbb{N} \rightarrow \mathbb{Z}; \)
let \( \text{zebra: } \mathbb{Z}; \)

\[
c(10) \text{ where } c(n) = d(n) + t \text{ where } t = (r * 100 \text{ where } r = 2) \\
\text{and } d(s) = (e(s) \text{ where } k = 10) \\
\text{and } e(r) = 20 + r + f(r) + \text{zebra} \\
\text{and } f(r) = \text{if } r = 0 \text{ then } 100 \text{ else } f(r - 1) \text{ endif} \\
\text{and } \text{zebra} = t
\]

-- Test for Multiple ands, multiple wheres and formal/variable collisions

-- Test for if-then-else collisions with multiple ands, wheres

let \( c: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( d: \mathbb{Z} \rightarrow \mathbb{Z}; \)
let \( e: \mathbb{Z} \rightarrow \mathbb{Z}; \)

164
c(5) where \( c(n) = d(n) + 12 \) where \( t = 100 \) and \( d(s) = (10 + e(s)) \) where \( k = 100 \) and \( e(k) = 20 + k + t \)

--Note the use of parenthesis here: if they are removed, the program will bomb because \( t \) will be undefined

ERROR TESTING

let \( x:Sz \)
let \( j:SZ \)
let \( i:Sz \)

where \( t = x \% j \)
and \( x = 5 \) and \( j = 0 \)

-- Gives Division by Zero run time error

let \( b:b \)
let \( i:SZ \)
let \( j:sz \)
let \( n:Sn \)
let \( x:sz \)

if \( b \) then \( i \)
elsif \(-b \land b \) then \( j \)
else \( x \) endif where

\( b = 1 = 2 \) where
\( i = 0 \)
and where \( j \)
and where \( z = 69 \)

-- Gives two parser errors line 13 and 14, \( j \) undefined and
-- where following \( \land \) and

let \( fac :SN > Sn \)

\( fac(5) \) where \( fac(n) = fac(n - 1) \)

-- Check for stack overflow

too much where \( too \ much = 1(0000) * 1(0000) \)

Check for Multiplication Overflow

too much where \( too \ much = 10000 + 10000 \)

Check for Addition overflow
too_much where too_much == -30000 - 30000

-- Check for Subtraction Overflow

let c : $Z -> $B;
let d : $Z -> $B;
let k : $Z -> $Z;
let g : $Z -> $Z;

c(1) \land d(2) where
d(n) == (1 = k(n - 1)) where
  k(1) == 1 + 10) and
c(n) == n = 3

-- Test for proper use of comments; note that there is no
delimiter on the second line of comments, as there should
-- be

MISCELLANEOUS TESTS

let b : $b;
let i : $Z;
let j : $Z;
let n : $n;
let x : $z;

if (b \lor -b) then i;
elsif (b \lor -b) then j
else x endif where
  b == 1=2 where
    i == 6
    and j == 2
    and x == 69
Test for not construct, boolean constructs

let h : $h;
let i : $Z;
let j : $z;
let n : $n;
let x : $z;

if -(b \lor -b) then i;
elsif -(b \land -b) then j
else x endif where
  b == 1=2 where
    i == 6
    and j == 2
    and x == 69
-- should give 2
-- Check and, or, notand, notor
-- Check if, else, elseif
-- Especially, check all in combination

let a:SZ;
let b:sz;
let y:sn;
let x:sz;
let f: $n*$n->$n;
let times : $n*$n->$n;

f(30,30) where
  f(a,b) == times(a,b) where
    times(x,y) == x*y
-- Multiargument Checking
-- Natural Type Checking

let a:SZ;
let b:sz;
let y:sz;
let x:sz;
let f: $z*$z->$z;
let times : $n*$n->$n;

f(30,4) where
  f(a,b) == times(a,b) where
    times(x,y) ==
      if (1 = 1) then x%y
      else 2 endif end
-- Integer Division Checking

let c : $Z -> $B;
let d : $Z -> $B;
let k : $Z -> $Z;
let g : $Z -> $Z;

c(1) ∧ d(2) where
  d (n) == (1 = k (n - 1) where
    k (l) == 1 + 10) and
  c (n) == n = 3
-- Test for proper use of "and" and implementation of
-- Parens
APPENDIX O – ROCK COMPILER USER’S MANUAL

I. Installation

The rock compiler program comes on a 5.25" disk with all public domain programs necessary to run it. To install this program on another floppy disk or a hard disk, use the following procedures:

1) Change the system drive to the disk drive containing the floppy disk.

2) Type "INSTALL", followed by a space and the drive and directory on which you want the program installed.

Note that the Rock compiler uses three unsupplied files to operate: RASM86, LINK86, and your choice of word processor. The RASM86 and LINK86 programs must be installed on the same directory as the compiler.

II. Running the Compiler

a. Type in "ROCK" and wait for the screen display shown in figure 1 to appear.

![ROCK Compiler Program](image)

Figure 1

b. When the prompt appears, type in the file name of the source file you want to compile, then press return. The compiler will accept directory specifications in the file designation. If the source file is found, the compilation will begin immediately, and the screen will appear as shown in figure 2. If the file is not found, the screen will appear as shown in figure 3.
c. If a successful compilation takes place, the prompt for a source file reappears. If the compilation is not successful, error messages will appear on the screen, and a copy of these messages can be found in a file named Errors.Phi. A typical error display is shown in figure 4. After perusing the errors, you may press any key to return to the prompt for a source file.
ROCKY ERRORS

line 1: forms list missing or error in forms list
line 1: misplaced or missing =
number of errors = 2

PRESS ANY KEY TO CONTINUE

Figure 4

d. If compilation is successful, both an .exe and an .obj file will be created. In the event that an error occurs, neither file will be created.
WARNING: If you choose to compile two programs with the same prefix, ensure you save the first one before compiling the second one; otherwise, the second compilation will overwrite the output file of the first compilation.

e. To cleanly stop the compiler, press the ESCAPE key any time the system asks for an input. If you have started to compile a program and you need a "panic" exit, press "Control-Break". If you do this, the cursor will not reappear on the screen. However, you can get it back by running the ROCK program again and making a normal exit.

III. Error Handling

Errors are divided into two categories: those found during compilation and those found during run time. The following two sections list the errors messages from both categories which you might encounter. Each message includes a brief synopsis of what causes the error.

COMPILER ERRORS

<table>
<thead>
<tr>
<th>Message</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incomplete 1-&gt;</td>
<td>Either an 'I' or '=' was found where '1-&gt;' was expected</td>
</tr>
<tr>
<td>\ without following /</td>
<td>A single backslash was found where a logical or construct () was expected</td>
</tr>
<tr>
<td>logical OR is V</td>
<td></td>
</tr>
</tbody>
</table>

170
An incomplete type declaration was found.

An illegal constant was found; in this example, "3."

An unterminated literal was found, or a literal spanned more than one line.

A character with no meaning was found in the source file; '#', in this example.

The source program is too big for the host machine to compile.

An illegal statement follows the specified character: '*', in the example.

An illegal type definition follows the specified character: '*', in the example.

An unspecified error was found after LET, and the compiler is so completely sandbagged that it cannot recover.

A declaration, preceded by "LET", was not followed by a semicolon.

An invalid expression was found.

An expression defining a type was either missing or incorrect.

Formals were expected but not found, or formals were incompletely specified.

A PHI keyword or delimiter was expected or not found: ')' in the example.

TYPE found without an identifier.

Improper or no expression found following AND.
missing or invalid auxdef after keyword WHERE
Improper or no definition following WHERE

missing or misplaced closing paren in formals list
Formals found without closing parenthesis.

error in processing multiple Actuals
One actual was found, but an error was spotted in a subsequent actual.

missing literal after keyword FILE
FILE was found without a file-name being designated.

missing or invalid exp following KEYWORD
A keyword was spotted, but the following expression was illegal.

IF statement w/o ENDIF
No ENDIF to close off an IF statement.

error in formals preceding I->
"I->" found, but the formals list preceding it contained an error.

missing or invalid QualExp following COMMA operator
A list of elements was found with an illegal expression in it.

error in ArgBinding check QualExp or closing bracket
An improper expression in an argument binding was found, or the closing bracket on an argument binding was not found.

OZONE LEVEL I:
Unimplemented feature found. for 19.99 the feature can be implemented in 1999

NUMERIC VALUE EXPECTED
Non-numeric type found where a numeric type was expected.

NATURAL EXPECTED
Natural type was not found where it was expected.

INTEGER OR NATURAL EXPECTED
Either an integer or natural type is proper, but neither was found.

ERROR IN TUPLE DEFINITION
A tuple is improperly defined the source file used improper types or number of types in defining the tuple. This can also mean a single variable was improperly defined.

UNDEFINED VARIABLE IN AND SCOPE
An undefined variable was found in one of the two branches of an
in its scope.

**FUNCTION WITHOUT FUNCTION DEFINITION**
A function was defined without a declaration of its type and formals.

**FORMALS MISMATCHED**
Formals in a function definition are not the same in either type or number as those in the function's declaration.

**FUNCTION CALLED WITHOUT FUNCTION DEFINITION**
No function definition found for the function called.

**REAL NUMBER EXPECTED**
An incorrect type was found where a real number was expected.

**INVALID CONSTANT**
An invalid constant was found.

**BOOLEAN VALUE EXPECTED**
A boolean value was expected, but none was found.

**BOOLEAN OPERATOR EXPECTED**
A boolean operator was expected, but none was found.

**OUT OF RUN-TIME MEMORY SPACE**
Not enough space to accommodate the program during run-time.

**RUN-TIME ERRORS**

**DIVISION BY ZERO**
Division by zero attempted.

**MULTIPLICATION OVERFLOW**
A multiplication operation resulted in a numeric value outside the language limits.

**ADDITION OVERFLOW**
An addition operation resulted in a numeric value outside the language limits.

**SUBTRACTION OVERFLOW**
A subtraction operation resulted in a numeric value outside the language limits.

**STACK/VARIABLE SPACE CRASH**
The stack overwrote the variable space.
# INITIAL DISTRIBUTION LIST

<table>
<thead>
<tr>
<th>No. Copies</th>
<th>No.</th>
<th>Name and Address</th>
</tr>
</thead>
</table>
| 2          | 1.  | Library, Code Ø142  
             Naval Postgraduate School  
             Monterey, California 93943–5002 |
| 2          | 2.  | Chairman, Code 52  
             Department of Computer Science  
             Naval Postgraduate School  
             Monterey, California 93943 |
| 5          | 3.  | Computer Technology Programs, Code 37  
             Naval Postgraduate School  
             Monterey, California 93943 |
| 1          | 4.  | D. L. Davis, Code 52Dv  
             Department of Computer Science  
             Naval Postgraduate School  
             Monterey, California 93943 |
| 1          | 5.  | B. J. MacLennan, Code 52Ml  
             Department of Computer Science  
             Naval Postgraduate School  
             Monterey, California 93943 |
| 3          | 6.  | Capt. J. E. Connell, USMC  
             #5 Jack Ray Park  
             Wentzville, Missouri 63385 |
| 3          | 7.  | Maj. E. J. Cole, USMC  
             156 Haviland Rd  
             Ridgefield, Connecticut Ø6877 |
| 2          | 8.  | Defense Technical Information Center  
             Cameron Station  
             Alexandria, Virginia 223Ø4–6145 |
END
9-87
DTIC