Small: A Contribution to the Mobile Programming System

by

Charles Randyl Britten

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

University of Washington

1982

Approved by _____
Hellmut Golde

Program Authorized
to Offer Degree _____

Date _____

Master’s Thesis

In presenting this thesis in partial fulfillment of the requirements for a Master of Science degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with “fail use” as prescribed in the U. S. Copyright Law. Any other reproduction for any purpose or by any means shall not be allowed without my written permission.

Signature _____

Date _____

Table of Contents

List of Figures

List of Tables

Acknowledgments

Small has been a project built on the shoulders of STAGE2, the Mobile Programming System by Bill Waite. Professor Waite's dedication to machine independent techniques have had the deepest influence on me from the time I was an Undergraduate at the University of Colorado in 1967. My deepest thanks are due to him, for without his inspiration my contribution would never have been.

Many others have contributed to the work contained herein. First my thanks to Mike Brengel for many discussions about language issues when we were systems programmers in the Atmospheric Sciences Department. Special thanks to Dick Curtiss who has generated more code in Small than I have. His careful efforts uncovered program bugs that I would have missed. Dick also wrote much of the machine dependent interface for the Intel 8080 version. And Thanks to Hoe Felsenstein who wrote Seattle Runoff in Pascal, which I converted to Small and used to format this thesis.

And finally my sincere appreciation to my advisor, Helmut Golde, especially for his infinite patience.

Chapter 1
Introduction

Small is a simple, mobile algorithmic language. This thesis describes the goals and design of the language, and its implementation by techniques that place a special emphasis on portability. Small is intended to be a practical language for portable software systems. As a language, Small has a simple set of features that are adequate for its intended purpose, and at the same time it is designed for growth toward a family of languages with more advanced features.

The portability of Small is achieved by using the STAGE2 macro processor [23] as a bootstrap compiler. Since STAGE2 is highly portable and can be implemented by a full bootstrap in about two weeks, it is possible to implement Small by a full bootstrap in about three weeks. Due to the current widespread availability of minicomputers and microprocessors, a half bootstrap, utilizing some existing implementation of STAGE2, could reduce the time to implement Small on a new system to about one week.

STAGE2 is used as a bootstrap compiler by combining it with a set of macros that translate statements in Small to

the assembly language of the target computer. In order to adapt Small to a new target computer a set of machine dependent macros must be written. The macros have been designed and organized in a way that makes this adaptation a straightforward task, given familiarity with the target system. Indeed, it would typically take more time to become familiar with the characteristics of the new target machine than to rewrite the machine dependent macros.

Many influences have shaped the design and development of Small, particularly the author's experience with Fortran, PL/I, Pascal, and a variety of assembly languages, plus a desire for a truly machine independent tool for systems programming. There has been previous work with goals or results similar to those expressed here [2,6,13,20,22]. No attempt has been made to cite all such similar work. Instead, a few lines of research that have been directly influential have been traced.

Chapter 2

Motivation and Goals

The primary motivation behind Small is the desire for simplicity, portability, and a practical high level alternative to assembly language. The goal of simplicity finds expression in a parsimony of features and a straightforward approach to implementation.

Portability has been approached in two ways. First, the language includes only those features known to be reasonably machine independent, and excludes certain features, such as pointer variables, that are prone to the introduction of machine dependencies in a program. Secondly, the methodology for implementing Small has been chosen to enhance mobility and minimize the work an implementer must do to transport Small to another computer system.

Another important goal is to allow the expression of algorithms using structured programming, and at the same time to provide methods of generating reasonably efficient code for typical computers. By doing this, Small becomes a practical alternative for many systems programming applications that traditionally have been done in assembly

language. Systems programming applications suitable for Small include editors, assemblers, compilers, loaders and library managers, and many of the software components that make up an operating system.

Language Design

Small is a high level programming language with a limited number of conventional features. A high level language may be viewed as an interface between the human world of algorithms and the machine world of logic circuits. One way to bridge the interface is for human beings to adapt to the machine world by dealing directly with the computer hardware mechanisms. But this has been found to be an inconvenient, tedious, and error prone approach even with the best of assembly language tools. The best advice that can be found on this subject suggests the use of a high level language whenever possible, and assembly language only when necessary.

The human side of the interface requires a high level language to be both readable and writable. To be writable it must be easy to learn and use, and must be reasonably concise. To be readable the meaning and effect of every language construct must be readily understandable and

unambiguous. To accomplish this, algorithms and programs are expressed in terms of words, symbols, forms, and data structures that are well known from the fields of mathematics, symbolic logic, and structured programming. Unfortunately, the number of useful and interesting language features seems to be without limit, and it is a constant temptation to include too much in a language. The design of Small avoids this by observing a strict rule of adopting only those features that can be justified as fundamental and necessary for the limited goals of the project. Every feature has been weighed against these goals, particularly the goals of simplicity, ease of implementation, and machine independence.

The machine side of the interface requires a translation that makes use of computing resources with acceptable efficiency. Efficiency is an issue that must be considered from several points of view. At compile-time the translation process should be acceptably fast and operate within available resources. It is not strictly necessary that the translation process be the fastest possible or use minimum resources as long as it is acceptable and meets the other goals. The compiled code generated by the translation must meet a more stringent test in order to be considered a practical alternative to assembly language.

The generation of reasonably efficient code can be guided by a concept known as a Knuth profile [15]. A Knuth profile is a plot of the frequency of use of various language statements versus the complexity of the statements. Such a plot shows that some of the simplest statements are used most frequently, and other more complex statements with decreasing frequency. The conclusion to be drawn is that the simple, frequently used statements are the best prospects for optimized code generation [26]. The more complex statements are of less urgency for optimization as long as the code generated for them is acceptable.

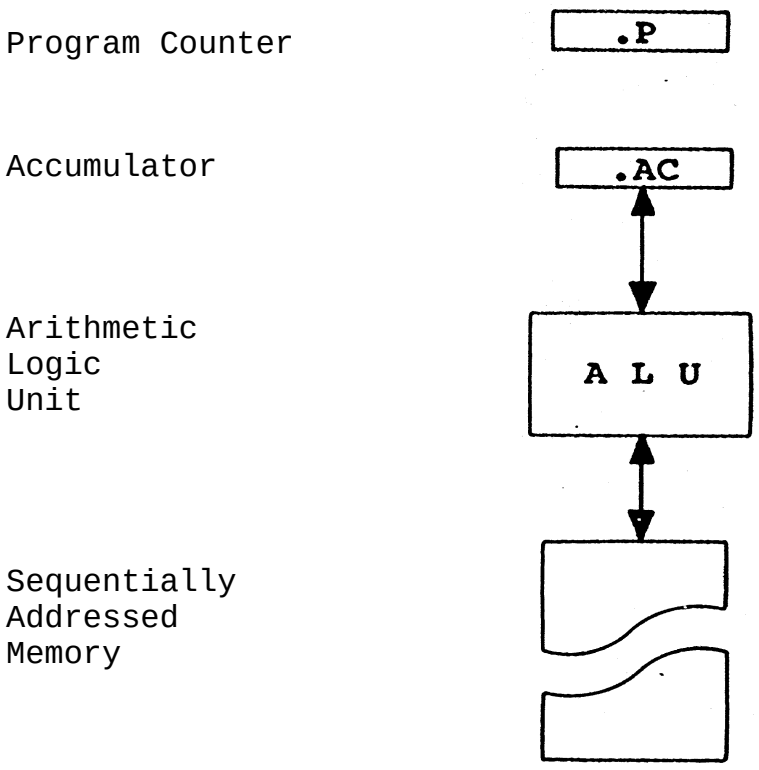
Abstract Machine Model

Underlying the design of Small is an abstract machine model. The purpose of the abstract machine model is to capture precisely the semantics of the language and to provide a well defined intermediate step in the translation of the language to actual machine code. It is not the purpose of the abstract machine model to represent an ideal computer which hardware builders should try to capture in their designs. The point of view that has been adopted here is that of natural science, that is to examine the varied population of existing computer designs for their

common elements, particularly those elements that can be accommodated to the goals of a simple language such as Small.

The abstract machine model is an interface between the high level language and the realization of the language on actual hardware. The essence of the abstract machine model is captured in the schematic diagram of figure 1, and in its abstract machine language, called MILL (Machine Independent Low Level), described in chapter 5.

The single accumulator model was chosen because it is close to the design of many actual computers. Another possible model would be a stack machine, such as the P-code model of many Pascal implementations. However, the use of a stack machine model would require more work for the implementer to generate code for typical computers, and is not as efficient a model as desired [18]. MILL code, based on the single accumulator model, maps to instructions for machines with one accumulator in a straightforward manner. Machines with several general registers can use mill code, in a first implementation, by selecting one of the general registers to correspond to the abstract accumulator. A more efficient implementation for a multiple register machine would require the addition of a register allocation algorithm. MILL code also maps to stack machine



Abstract Machine Model

Figure 1

architectures in a straightforward way, and results in minimum stack depth.

No input or output is shown in the abstract model. It is assumed that Small and the abstract model are to be implemented by imbedding in an operating system, and that input and output are provided by calls to the operating system. Small adopts certain conventions for input and output as described in chapter 5.

Modular Programming

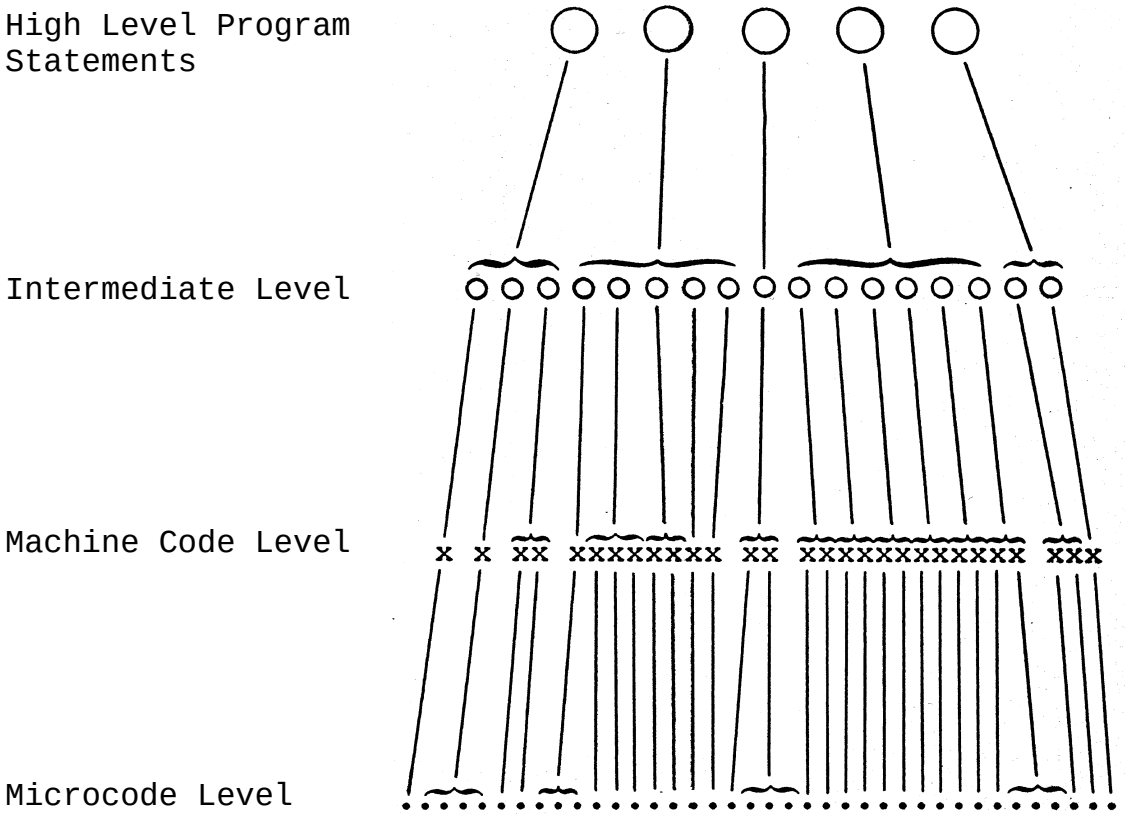
Experience with large programming projects in industry led to the conclusion that separately compiled modules were essential as a practical matter. Modular programming permits local changes to be made without necessitating a complete recompilation of a large program. The ability to combine independently compiled modules helps foster the orderly evolution of a programming project by focusing attention on the definition of simple interfaces between modules, and isolating low level implementation details within modules.

Chapter 3
Historical Background

Professor Maurice V. Wilkes has been recognized as the father of microprogramming. It is less widely recognized that there is a direct relationship between microprogramming, macro processing, and Wilkes's early work. Macro processing and microprogramming both may be viewed as one-to-many mappings of the primitive elements of one level to the primitive elements of another lower level in the process of executing a high level language statement. The fact that the two processes are related as segments of a single continuum [7] is illustrated in figure 2.

WISP

In 1964 Professor Wilkes described a simple list processing language called WISP [25]. WISP began as a student exercise on the EDSAC (Electronic Delay Storage Automatic Calculator) computer at the Mathematical Laboratory of the University of Cambridge. The EDSAC was one of the first computers that became known as von Neumann machines. Appendix A contains a brief account of the



Levels of Translation and Execution

Figure 2

EDSAC.

WISP was used as an experiment in machine independent compiling. The language and its implementation were developed as a sequence of bootstrapping steps beginning with an extremely simple compiler called TR0. As a language, WISP is like an assembly language for a LISP machine, with simple statements to manipulate and test CAR and CDR fields representing the left and right halves of linked list elements. The CDR field always contains the address of another element or a null (end of list mark), and the CAR field may contain a character code, an address, or a null.

The TR0 compiler consisted of 74 lines, written in a simple version of WISP, that was translated by hand and combined with a set of machine code subroutines to produce the first working compiler. This compiler was used to compile TR0.1, and 84 line compiler written in a subset of WISP with 24 statement types. Next Wilkes described TR1, a 180 line version that added nested conditional statements, recursion, and access to machine code subroutines to the language. This expanded language was used to write the next compiler, TR2, in 262 lines, many of which contained several statements.

Each of the WISP compilers was implemented as a macro processor. The macro processor operation began by reading a set of statement templates aand machine code translations that defined the language to be compiled. The language definition phase was followed by a source program to be translated. Each statement of the source program was matched against the set of templates, and when a match was found the corresponding machine code translation was produced together with any parameters that were to be substituted into the output. A modern version of WISP is included in Waite's textbook on non-numeric processing [23]. His version is implemented with a special set of macros.

WISP was successful as a machine independent experiment and was transported to an Elliot 803, an IBM 709, and later to an IBM 7094 at Columbia University and many other computers typical of the sixties. It was used to implement symbolic differentiation, an Autocode compiler, and an early assembler for the English Electric KDF9, as well as a series of self-compiling compilers for WISP. In the late sixties a document on WISP was available at the University of Colorado [19] that described a version with 39 statement types and a compiler written in 542 lines called BASCMP (basic compiler). BASCMP was a fairly

general macro processor that was capable of compiling a rich version of the WISP language with 74 statement types that made it suitable for a variety of applications.

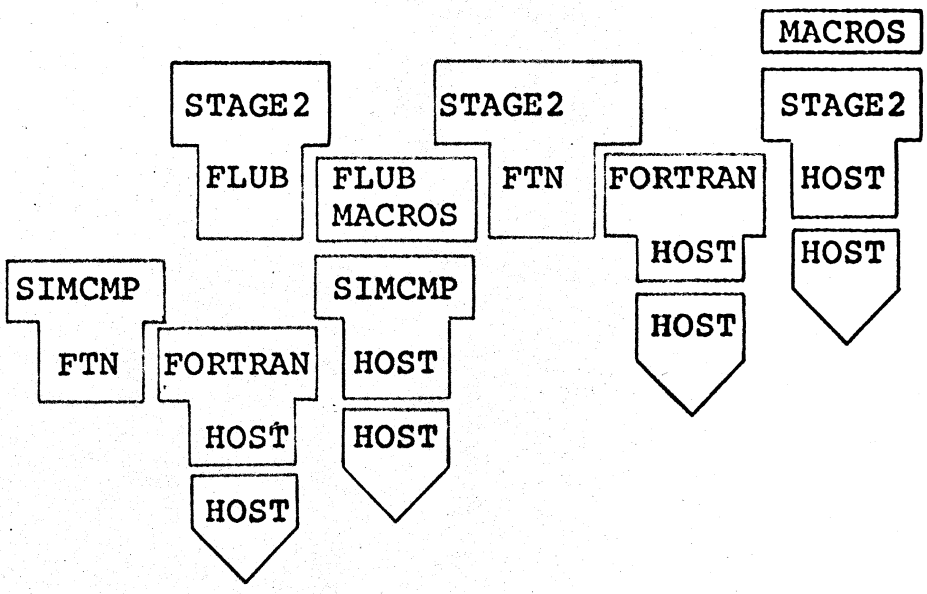
WISP is interesting for its place in programming language history as an early portable language with a portable implementation methodology. Wisp was one of the first examples of a language implemented by a self-compiling compiler. Its simplicity and its informal style have been a source of inspiration to me since I was first exposed to it. I have included this historical review of it here for that reason.

STAGE2

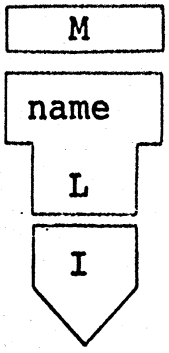
The STAGE2 macro processor is fundamental to the existence of Small, since STAGE2, combined with a set of macros, is employed as the bootstrap compiler for Small. STAGE2 was written by Professor William M. Waite of the University of Colorado and is completely documented, including listings and implementation notes, in his textbook on non-numeric processing [23]. An early version of STAGE2 was available at the University of Colorado in the late sixties.

STAGE2 is so named because it is the second stage in a bootstrapping sequence that begins with a simple compiler called SIMCMP. The SIMCMP algorithm is expressed as a 76 line Fortran program that is also described and included in Waite's book. Fortran is often convenient, but is not essential, for the implementation and use of SIMCMP. It is a simple matter to translate SIMCMP by hand to some other language, even an assembly language. The SIMCMP algorithm is directly related to the TR0 and TR1 bootstrap compilers of WISP.

Figure 3 illustrates the implementation of STAGE2 by a full bootstrap using Fortran and SIMCMP. Each "T" in the diagram represents a translator, where the name of the translator is written in the crossbar, and the native language is written in the stem of the "T." Each translator converts the object on its left to the object on its right, where "FTN" denotes Fortran as a native language, and "HOST" denotes the machine language of a suitable host computer. Where the translator is a macro processor, a set of macros is required to define a particular translation. A set of macros is shown as a rectangle above the "T" diagram of a macro processor.



Legend:



A Translator

name – name of translator

L – language or machine in which translator is expressed

I – an interpreter or machine that executes the program immediately above it

M – a set of macros or tables that configures the translator below it

A Bootstrap Implementation of STAGE2

Figure 3

SIMCMP operates by reading a language definition, consisting of a set of macro templates and code bodies, and storing the macros in a linked list data structure. Then the statements of a source program are read and matched against the templates, and the corresponding macro code bodies, acting as rewrite rules, generate a stream of output text. SIMCMP is used to translate the language in which STAGE2 is written to some other language suitable for a first implementation of STAGE2. Waite's documentation contains instructions and listings for translating the source for STAGE2 into Fortran. It is worth pointing out that Fortran may be convenient in this instance also, but that Fortran is not essential for the implementation of STAGE2. It is a simple matter to write a set of macros to generate some other language, even an assembly language, as the translation. SIMCMP is too primitive to generate optimized assembly language, but this is not important since the only purpose of SIMCMP is to generate the first working version of STAGE2. Once a version of STAGE2 is operating, even if in a non-optimal form, it can be used to retranslate the STAGE2 source, typically to the assembly language of the target machine, with optimizations suitable for satisfactory operational performance.

Janus

The MILL intermediate language of Small has been influenced by the work on Janus by Coleman, Haddon, Poole, and Waite [8,9,12]. Janus, however, attempts to solve the much more difficult “m by n” problem in which m different languages are to be implemented on n different machines. It addresses the problem by defining Janus as a universal intermediate language that interfaces between the n machines and m languages. A compiler using this technique would be implemented in two phases. First, statements in the subject language would be translated to Janus by a language dependent phase. And second, the Janus statements would be translated to target machine code by a machine dependent phase. The design of the primitives of Janus emphasizes what the statements of the subject language mean, as opposed to how actions are to be carried out. The primary difference between Janus and MILL is that MILL is much simpler since it has fewer, less difficult requirements to meet. A second fundamental difference is that MILL is based on a single accumulator abstract model, whereas Janus in based on a stack machine model.

Chapter 4
The STAGE2 Implementation

Small is implemented by supplying STAGE2 with a set of macros that translate statements in Small to a target machine assembly language. The macros consist of a set of machine independent nucleus macros, and a set of macros that depend on the target machine. The nucleus macros and the machine dependent macros are combined in a single set that performs a one pass translation to be followed by the target machine assembler. The translation produces two output files. One is a listing file of the source program with line numbers, level numbers, and an error summary added. The other output is the assembly language translation, with the original source statements converted to comments to the assembler [5].

It is beyond the scope of this thesis to explain the syntax of STAGE2 macros, however a few notes are in order to help make the examples understandable. STAGE2 is, in effect, a terse language for recursive string manipulation. Full details, including explanatory examples, are given in Waite's text [23].

In the examples in figures 4 and 5, "#" is an end of

line marker, and a “#” on a line by itself is the end of a macro. The first line of a macro is a template in which a “$” is a parameter flag that matches a string in the source line with rules similar to ARB in SNOBOL4 [11]. All other characters in a template must match exactly. The “$10” in a code body means “produce a copy of parameter 1 here,” where parameter 1 is the string that matches the first “$” in the template. The “0” in “$10” means perform conversion 0 on parameter 1, which leaves it unchanged. The notation “$20” refers in a similar way to the second parameter. Then “$F13” invokes STAGE2 built-in function 1 to output the constructed line to channel 3, which happens to be the file for the assembly language output. If a code body line is not sent to the output by a function such as “$F13” it is meant to match another macro template.

Appendix C contains a complete listing of the nucleus macros that translate Small to the MILL abstract machine language.

The nucleus macros are almost totally machine independent. There are, in fact, two minor changes that may have to be made to the nucleus macros to adapt to a new target machine. The first change would be required to generate comments compatible with the target assembler. Some assemblers, for example, recognize comments beginning

with an asterisk, others with a semicolon or other character, and still other assemblers have other rules for comments. The following figure shows two possible versions of the .INT macro that handles all comments. The .INT macro is second from the last in the listing in Appendix C.

```
.INT $#          All nucleus comments processed here
*$10$F13#          Generate assembler comment with *
#                  Output to STAGE2 channel 3

.INT $#          All nucleus comments processed here
;$10$F13#          Generate assembler comment with ;
#                  Output to STAGE2 channel 3
```

Two Versions of the Nucleus Macro for
Target Machine Assembler Comments

Figure 4

The other change to the nucleus macros has to do with array element addressing. The only data type in Small is the integer word. Since integer words are used to calculate addresses the word size must be large enough to hold an address. This implies that the word size must be at least 16 bits, since an address space limited by a smaller word size would be too restricted to be useful.

On a minicomputer with 16 bit words, in which each consecutive word has an address differing from its neighbor by one, there would be a one-to-one correspondence between the integer words of Small and the 16 bit machine words. Such a machine would be said to have one address unit per

word.

On a computer in which each 8 bit byte has a unique address, the integer words of Small would again have 16 bits, but in this case would have two address units per word. On such machines, the address of an array element must be calculated by multiplying the index be two before adding the array base address. Of course, such a multiplication can be done by a left shift. The following figure provides two versions of nucleus macro 76, which evaluates array element accesses.

```
.VAR $($)#                Array element with
.EXP, =$10+($20) SHL 1#     two address units
# 76                        per word

.VAR $($)#                Array element with
.EXP, =$10+($20)#           one address unit
# 76                        per word
```

Array Element Address Macro

Figure 5

The present version of the nucleus macros consists of 719 lines of macro code for STAGE2. The machine dependent macros for the Intel 8080 microprocessor consist of 233 lines which replace the last eight lines of the nucleus macros when the two files are combined. The result is a file of 944 lines that translates Small to 8080 assembly

language in a format acceptable to the Microsoft MACRO-80 assembler or the Intel ASM-80 assembler. Both assemblers produce relocatable object code modules that can be combined by a suitable linking loader. The Microsoft LINK-80 loader has an option to generate a symbol table file containing the names and locations of all global variables. This file can be used by the SID * (Symbolic Instruction Debugger) program that is available from Digital Research with their CP/M * operating system. The use of CP/M, the Microsoft assembler and loader, and SID rounds out an environment for implementing and testing programs written in Small on a typical microcomputer.

Compiler Design

A computer may be viewed as a set of filters that transform an input stream of symbols to an output stream by a set of rewrite rules. There may be intermediate queues and stacks where some symbols are stored until some event occurs, such as the end of an expression, that permits the queued or stacked symbols to be processed by another set of filters on their way to the output stream.

---

* CP/M and SID are registered trademarks of Digital Research, Inc.

There are also side effects of certain symbols that influence the way later symbols are processed. For example, the instructions generated to access a variable will be different depending on whether the variable is the name of a simple storage location or a formal parameter in a procedure. The declarations that define a variable must come before the expressions that use it, and must save the attributes that affect how the variable will be accessed.

The view of a compiler as a set of filters works well with the implementation of Small by STAGE2 macros. The macros are organized into subsets, and each subset performs a transforming function that moves the stream of symbols closer to the form desired for the output. For example, there is a set of macros to strip leading blanks, trailing blanks, and comments, and to break up the input stream into normalized syntax units. Then there is a set of macros that recognizes the various syntax units. Syntax units are parts of statements that are identified by a keyword or a special symbol. For example, the IF statement consists of four syntax units: (1) IF cexp; (2) THEN stmt; (3) ELSE stmt; (4) ENDIF. Many syntax unit processors are recursive since each “stmt” may consist of one or more statements including, for example, complete nested IF statements. The only syntax unit that does not begin with a keyword is the one for assignment statements, which are recognized by an

equal sign. The technique is, in effect, recursive descent by macro template matching.

The notion of compiling by a set of filters is carried further by applying it to the recursive descent processing of infix expressions. Repressions are broken up into terms by a set of macros that recognize the additive operators: +, -, OR, and XOR. The terms are broken into primaries by a set of macros that recognize the multiplicative operators: *, /, MOD, AND, SHL, and SHR. The primaries are processed by a set of macros that recognize integer constants, identifiers, array elements or function calls, and subexpressions enclosed in parentheses. The net result of the expression processing filters is a postfix queue containing a string of operators and operands in postfix order representing the original expression.

The postfix queue is processed by a set of code generation macros that produce MILL code. With the addition of a set of machine dependent macros, MILL code is then translated to target machine code.

Chapter 5

Language Elements

Small is designed to include a set of language elements that are adequate for systems programming tasks and can be implemented by the bootstrapping technique described in previous chapters. The language elements include the flow of control forms of structured programming, procedures with optional recursion, declarations for scalar variables and constants, arrays with optional initialization, compile-time variables useful for parameterized declarations, and separately compiled modules. Each language element is described in this chapter, which serves as the language reference manual. Appendix B contains syntax diagrams for the language.

Flow of Control

The classical flow of control forms of structured programming, as implemented in Small, are as follows:

1) IF condition;
   THEN statements
   ENDIF

2) DO WHILE condition;
   statements
   ENDDO

3) sequential execution of statements

The use of an explicit terminating keyword, such as ENDIF, for each of the flow of control statements means that there is no need for redundant bracket words, such as BEGIN and END, to delimit a block of statements.

Although the forms above are known to be sufficient for the expression of algorithms, many algorithmic languages include additional forms for the sake of programming convenience and efficiency [4,10]. Small also adopts this policy, and as a result includes the following additional forms. These are considered justified because they significantly add to the utility of the language at a nominal cost.

```
4)  IF conditional;
      THEN statements
      ELSE statements
      ENDIF

5)  REPEAT statements
      UNTIL condition;

6)  CASE expression;
      OF case-label-list;
      set-of-cases
      ENDCASE
```

Small also includes a GO TO statements and a LABEL statement. Although the use of unnecessary labels and GO TO statements is to be discouraged, it can be shown that

certain algorithms cannot be expressed most efficiently without their use [16]. In such cases their use is considered justified. The GO TO statement transfers control to the first statement after a LABEL with the same name.

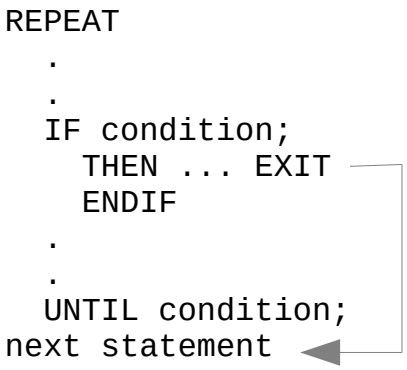
7) GO TO name

8) LABEL name

The final flow of control feature is the EXIT statement. It has the effect of a controlled GO TO which passes control to the next statement following a surrounding control structure.

9) EXIT

An EXIT statement normally makes sense only as the last statement in a THEN or ELSE clause. Its semantics is best illustrated by an example.



Procedures

Small includes both recursive and non-recursive procedures. It can be observed that there is often a significant additional cost in memory utilization and execution time for the implementation of recursion on existing computers such as the IBM 360, CDC 6000 and many minicomputers. Small minimizes the overhead associated with recursion by assuming that only the return address is stacked on entry to a recursive procedure. The non-recursive procedures drop the assumption of a stacked return address, and permit the return address to be accessed by the most efficient mechanism provided by the target hardware.

```
10)  RECURSIVE PROCEDURE name[(arg-list)]
       statements
       ENDPROC

11)  PROCEDURE name[(arg-list)]
       statements
       ENDPROC
```

NOTE: The metasyntactic brackets, “[“ and “]”, indicate that the enclosed items are optional.

Since only the return address is saved on the stack, it is the programmer’s responsibility to create any local stacks needed to save local variables or parameters when using recursion. This requirement is made for the sake of

simplicity of implementation, and to avoid having mechanisms hidden from the user.

A procedure normally contains at least one RETURN statement. If the procedure is recursive the return uses the address currently on the return address stack. A scalar value may be returned from a procedure called as a function.

12) RETURN

13) RETURN expression;

As mentioned previously, Small attempts to deal with the population of existing computers from the point of view of natural science. It can be observed that there are many computers, such as the PDP-11 and Intel 8080, that provide for a return address stack in their architectural designs. Such machines may support recursion as their natural mode of operation, and the hardware call and return mechanisms may be sufficiently efficient that non-recursive procedures actually use the same mechanisms.

There are also many machines, such as the CDC 6000 series, the Data General Nova, Prime, and Honeywell minicomputers, and the Texas Instruments TMS 9900 microcomputer, that do not directly provide a return

address stack mechanism. From a natural science point of view we cannot simply dismiss these designs as obsolete. On this class of machines the return address stack must be implemented in software for recursive procedures, while non-recursive procedures may use the most efficient available hardware mechanisms.

For those machines that require a software implemented return address stack, Small provides a declaration to give the programmer control of the stack allocation.

```
14)  DCL RECURSIVE RETURN(stacksize);
```

This declaration automatically makes the return stack global so that the recursive procedures in other modules can make use of it. On those machines that implement the stack with hardware, such as the PDP-11, Intel 8080 and 8086, the stack space is assumed to be hidden from the programmer. On such machines a declaration such as 14) is not required and has no effect if it is included. It is part of the design philosophy of Small to eliminate hidden mechanisms whenever possible. Therefore if 14) can have meaning on the target machine it is assumed to be implemented as such, and not hidden from the programmer.

Procedures may be called by function calls within expressions, or by CALL statements. It is assumed that a

procedure called as a function returns a value to the containing expression, and that a procedure called by a CALL statement does not return a value.

15) name(arg-list)
(function call as primary in expression)

16) CALL name[(arg-list)]

It is assumed that the calling mechanism makes no distinction between recursive and non-recursive procedures, and that the differences, if any, are dealt with in the procedure entry and return mechanisms.

Example of a recursive procedure:

```
  DCL RECURSIVE RETURN(8);   * return address stack
  DCL STP=0,STK(8);          * local parameter stack
  DCL R,T;                   * local variables
*---------------  FACTORIAL EXAMPLE
  RECURSIVE PROCEDURE FACT(N);
    IF N LE 1;
      THEN RETURN 1;         * terminate the recursion
      ELSE
        STP=STP+1;
        STK(STP)=N;          * stack the parameter
        T=N-1;               * compute next parameter
        R=FACT(T)*STK(STP);  * recursive call
        STP=STP-1;
        RETURN R;            * return result
      ENDIF
    ENDPROC
```

Assignment Statements

In the present version of Small, assignment statements and expressions assume integer operators and operands, and produce integer results. However, Small is designed to

accommodate additional data types within the present framework.

17) variable=expression

An operand may be a scalar variable, an array element, a call on a function that produces an integer result, or a character code represented by a single quoted character. The operators on integers, in order of precedence, are as follows:

Unary Operators:	+ - NOT
Multiplicative Operators:	* / MOD AND SHL SHR
Additive Operators:	+ - OR XOR
Relational Operators:	EQ NE LE LT GE GT

The arithmetic operators produce signed integer results, and the logical operators act on all bit positions of the operand words. SHL and SHR are shift left and shift right operators respectively.

The relational operators are restricted to use in conditional expressions which appear in IF, DO WHILE, and REPEAT UNTIL statements. No more that one relational operator may be used in a conditional expression. The restrictions are due to the method of implementation. The evaluation of a conditional expression results in an integer value and a MILL conditional jump instruction. If the integer value is non-zero the condition is considered true, and if it is zero it is considered false. The

conditional jump is taken according to the value of the condition. There is no boolean type in the present version of Small.

Operating System Interface

The ENTRY and EXTERNAL statements declare global variables and procedure names for use by a suitable linking loader.

18) ENTRY ent-list;

19) EXTERNAL ext-list;

Before an external procedure can be used it must be declared as in the following example:

```
EXTERNAL PROCEDURE YONDER;
```

If a suite of modules are to use the same return address stack, each module containing a recursive procedure can gain access to the stack by:

```
EXTERNAL RECURSIVE RETURN;
```

Also see the return stack declaration 13).

The START statement is used to specify the label where

execution of the program is to start. It should be the first code generating statement in the first loaded module of a program, and would typically translate to a jump instruction. However, some operating systems require some special treatment for a starting location. The STOP statement returns control to the operating system.

```
20)  START label
21)  STOP
```

Each module begins with a BEGIN statement that may have an optional module name. The module name generally is not required but may be used to identify the name of the file that contains the module. Each module ends with and END statement.

```
22)  BEGIN [modulename]
23)  END
```

The BEGIN statement must be the first statement of a module, not counting comments. Small does not support nested scopes for identifiers, and nested BEGIN END brackets are not allowed. All identifiers within a module are global within that module.

Input and output are system dependent, and it is assumed that Small is to be implemented on systems that provide a host file system. Files are assumed to consist

of lines of characters terminated by carriage returns. By convention, READ and WRITE are procedures callable from Small that provide the interface to the host file system.

READ and WRITE each take two arguments. The first argument is a unit number, typically from one through nine, that is equivalent to a Fortran unit number. The second argument is an array that represents a string of characters, one character per word. By convention the first element of the array, the element whose index is zero, is known as the length element and contains the current number of characters in the string.

READ is called as a function that returns a value indicating the status of the file. A retrun value of zero indicates end of file, one indicates that a valid line has been read, and a negative value indicates an error condition.

```
24)  status=READ(unit,line);
25)  CALL WRITE(unit,line)
```

READ gets an array of characters from the file, and sets the length of the line to the number of characters actually read. When READ encounters end of file the length is set to zero. An end of file is written by setting the length to zero before the call to WRITE.

Example of READ and WRITE:

```
* Copy IN to OUT until end of file
  DCL IN=1,OUT=2,STATUS;
  DCL EOF=0,LINE(80);
*---------------
  PROCEDURE COPY;
    REPEAT
      STATUS=READ(IN,LINE);
      CALL WRITE(OUT,LINE);
      UNTIL STATUS EQ EOF;
    RETURN
    ENDPROC
```

Conversion between integers and strings of digits is easily written in Small. As examples of such routines, IREAD, IFORM, and CAT2 are given in Appendix D.

Declarations

Small provides the ability to set a compile-time parameter with a SET statement. Such parameter declarations are useful for controlling generalized data structures.

26) SET name=pexp;

The pexp is a parameter expression that is evaluated at compile-time. A parameter expression may consist of arithmetic operators, integers, and any parameters set previously. In the STAGE2 implementation, the pexp is evaluated by the STAGE2 built-in expression evaluator.

All variables must be declared before they are used in Small. A declaration causes definition of a label and static allocation of space at the point of appearance. A scalar variable may be declared with an optional initialization value. The keyword DECLARE may be abbreviated to DCL.

```
27)  DCL name;          * scalar variable
28)  DCL name=pexp;     * initialize to pexp value
29)  DCL name='char';   * initialize to char code
```

A declaration statement may contain several items separated by commas.

One-dimensional arrays may be declared with optional initialization.

```
30)  DCL name(pexp);     * array of size pexp
31)  DCL name=(datalist);
```

An array declared by name(pexp) will occupy pexp+1 words and can be indexed by zero through the value of pexp. The datalist initialization may contain constants, repeated constants, and repeated structures of constants. Repetition is indicated by an integer, followed by an asterisk, followed by a data element or structure to be repeated. For example:

```
DCL ARRAY=(1,2*(3,2*1));       * these are
DCL ARRAY=(1,3,1,1,3,1,1);     * equivalent
```

A string of character codes can be declared by a MESSAGE declaration.

```
32)  MESSAGE name='any string';
```

This declaration creates an array of n+1 words with the first element containing the length n of the string, and the following words containing one character code per word. In the STAGE2 implementation the string may not contain an apostrophe.

Comments

A comment is indicated by an asterisk at any point where a statement could begin, and extends to the end of the line.

```
33)  * any comment to the end of the line
```

This form of comment is efficient because it need not be scanned for an end of comment symbol. The scan stops at the asterisk and resumes with the first character of the next line. It is also easy to write, requiring only one

character to begin a comment, and easy to read, since each comment is immediately and unambiguously identified as such. It is not possible to embed comments in expressions. Consequently there is no ambiguity between the asterisk as a comment symbol and the multiplication operator.

Abbreviations

Small provides abbreviations similar to those of PL/I for several of the keywords. It has been found that after the full spelling has been used once or twice for the sake of readability, the abbreviation will tend to be used thereafter for the sake of writability and brevity.

Keyword	Abbreviation
DECLARE	DCL
ENTRY	ENT
EXTERNAL	EXT
MESSAGE	MSG
PROCEDURE	PROC
RECURSIVE	REC

Character Set

A character set of 48 characters listed below is required for programs written in Small. The use of a restricted character set promotes portability to systems

with limited peripheral devices. The use of other characters is implementation dependent and may compromise portability.

Digits:	0 1 2 3 4 5 6 7 8 9
Letters:	A B C D E F G H I J K L M
	N O P Q R S T U V W X Y Z
Operators:	+ - * / =
Punctuation:	. , ; ( ) ‘
	blank

The MILL Intermediate Language

The design of MILL is much simpler than that of Janus. MILL, like Janus, emphasizes what information is to be carried from the syntax analysis phase to the code generation phase, without overly specifying how certain semantic actions are to be carried out in machine code. MILL differs from Janus in being based on an abstract machine model with a single accumulator, whereas Janus is based on a stack machine model. This implies that MILL carries the translation of expressions a step closer to the level of most machines, since very few machines actually use a hardware implemented evaluation stack.

The MILL language is divided into instructions and

pseudo-operations as shown in tables 1 and 2. The MILL instructions typically map to machine instructions in a straightforward manner, however the pseudo-operations, such as those for a procedure call and entry sequence, often require special attention.

The arithmetic instructions operate on and produce signed integer words. The logical instructions operate on all bit positions of a word. Small is intended to require a minimum of run-tme support. However if a hardware instruction equivalent to a MILL instructions is not available it is sometimes best to use a short support routine to simulate the MILL instruction. Examples of support routines are the 16-bit multiply and divide routines needed by the Intel 8080 implementation. Otherwise, if only a few target machine instructions are needed to simulate the requires operation it is usually best to generate the target instruction sequence in-line.

Table 1

MILL Instructions

Arithmetic and Logical Instructions:

L	opnd	Load operand into accumulator
+	opnd	Add operand to accumulator
-	opnd	Subtract operand from accumulator
*	opnd	Multiply accumulator by operand
/	opnd	Divide accumulator by operand
MOD	opnd	Remainder of accumulator/operand
AND	opnd	Bitwise AND accumulator with operand
OR	opnd	Bitwise OR accumulator with operand
XOR	opnd	Bitwise exclusive or accumulator with operand
SHL	opnd	Shift accumulator left by operand
SHR	opnd	Shift accumulator right by operand
-		Negate accumulator
NOT		Complement accumulator (invert bits)
ST	opnd	Store accumulator at operand
L	*.AC	Load indirect through accumulator
ST	*label	Store accumulator indirect using address at label

Jump Instructions:

J	label	Unconditional jump to label
JEQ	label	Jump if accumulator equal to zero
JNE	label	Jump if accumulator not zero
JLE	label	Jump if accumulator less than or equal to zero
JGT	label	Jump if accumulator greater than zero
JX	n	Indexed jump for case statement
JC	label	Case label pointer

Forms of “opnd” in Arithmetic and Logical Instructions:

label	Operand is a label
=n	Operand is integer n
=label	Operand is value of label
D name	Operand is formal parameter

Table 2

MILL Pseudo-Operations

BEGIN		Initialize module
STRT	label	Start execution at label
LABEL	label	Define label at present location
SPACE	n	Skip space for n words
CONST	n	Word containing constant n
ENT	label	Define label as a global entry
EXT	label	Define label as external
SUBR	label	Non-recursive procedure entry
RSUBR	label	Recursive procedure entry
NPARS	n	Number of formal parameters
PAR	name	Define formal parameter name
DEND		End of formal parameter list
ARGT	name,n	Transfer formal parameter name to corresponding actual parameter n
SCALL	label	Procedure call
NARGS	n	Number of actual parameters
ARG	label	Pointer to actual parameter
CEND		End of actual parameter list
RETN	label,n	Return from procedure with n parameters
RRETN	label,n	Recursive return from procedure with n parameters
END		End of module

Parameter Passing

Procedures are called as functions or with a CALL statement. In both cases parameters are passed using call by reference. Consider, for example:

```
P=Q(U,V,W);
CALL R(X,Y,Z)
```

The calling and parameter passing mechanisms are the same in each case. One possible algorithm to generate MILL code would be as follows:

(Algorithm P1)

1) Count the actual parameters
2) Generate

```
SCALL name     * pseudo-op to call the proc
```

3) Generate

```
NARGS  count   * number of actual parameters
```

4) For each actual parameter generate

```
ARG  name      * address of actual parameter
```

5) Generate

```
CEND           * end of calling sequence
```

For example:

Small	becomes	MILL
CALL R(X,Y,Z)		SCALL R NARGS 3 ARG X ARG Y ARG Z CEND

This abstract calling sequence is symmetrical with the corresponding procedure entry mechanism. For example:

PROC R(A,B,C)	becomes	SUBR R NAPRS 3 PAR A PAR B PAR C DEND

The intention behind these abstract mechanisms is to provide enough information to permit a systems programmer to define a straightforward mapping onto a typical machine instruction set. There are many possible mappings to actual hardware mechanisms. Consider, for example, the following typical implementations:

1) The ARG list could be copied to the PAR list by an argument transfer routine. This is one of the most straightforward techniques but is not necessarily the most efficient because of the movement of addresses, the duplicate storage, and the need for a run-time support routine. It is a reasonable technique for machines with primitive addressing modes. It is the technique used for the Intel 8080 and Prime computer

implementations.

2) The ARG list could be pushed on a stack by the calling sequence and popped into the PAR list by the procedure entry sequence. This is a very straightforward technique, but it may be even less efficient than implementation 1 if the pointers are moved from memory to memory twice, as they would be if the stack is in memory. This technique is often popular on machines with stack architectures.

3) If the target instruction set has indexing with offsets, it is possible to eliminate the PAR list in actual storage. Instead, an index register can be dedicated to point to the ARG list in the calling sequence. Then each formal parameter can be referred to by the offset in the ARG list using the dedicated index register as a base. This is an attractive technique because of the reduced overhead. Access to each formal parameter is reasonably efficient, assuming the hardware indexing mechanism is efficient. This technique is used in the implementation for the Intel 8086.

There is a problem with algorithm P1 in that it fails if any of the calling sequence parameters happens to be a formal parameter. The reason is that each ARG is presumed

to be a pointer to the actual storage location of the actual parameter. In order to resolve the problem the algorithm must be revised to evaluate each calling sequence parameter. Algorithm P2 solves the problem.

(Algorithm P2)

1) For each actual parameter
   increment count
   eval(param)
2) Generate
   SCALL name
3) Generate
   NARGS count
4) For each actual parameter generate
   ARG name
5) Generate
   CEND

The expression eval(param) must evaluate the calling parameter and take whatever action is necessary to ensure that the corresponding “ARG name” resolves as a pointer to the actual parameter storage location. If the argument of eval is not a formal parameter no further evaluation is needed. If the argument is a formal parameter the pointer to the corresponding actual parameter must be resolved. This can be indicated at the abstract MILL level by issuing a pseudo-op:

```
ARGT name,n          *actual pointer transfer
```

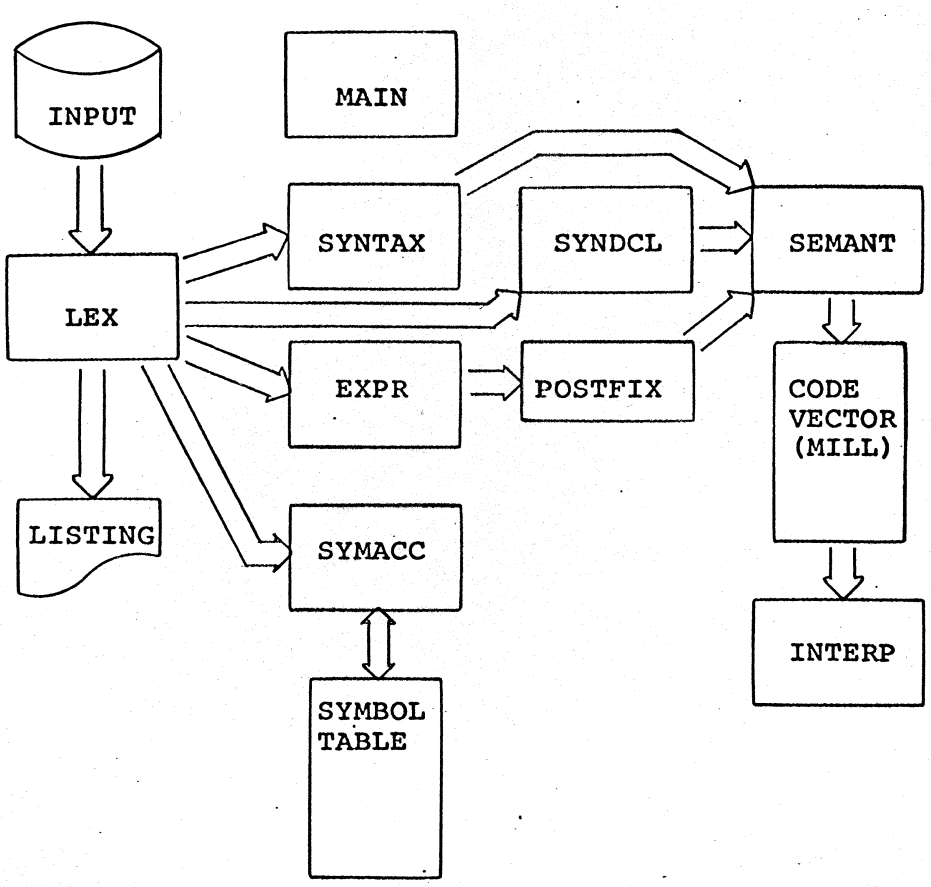
in place of eval(param). The implementer must translate the "ARGT name,n" pseudo-op into actual machine code that is compatible with the actual calling and parameter passing mechanisms that have been chosen. One possible translation of "ARGT name,n" would copy the address of the actual parameter to the corresponding "ARG name" in the calling sequence, where n is the number of the parameter. This is the technique used in the 8080 implementation.

Chapter 6
Implementation of a Compiler-Interpreter

In order to demonstrate the use of Small for practical systems software, the design and implementation of a compiler-interpreter is presented. A compiler-interpreter is a software tool that translates a high level language to an intermediate language that is executed by an interpreter. In the present case the source language is Small, and the intermediate language is MILL, represented in a form suitable for execution by an interpreter.

A compiler-interpreter is an attractive project for several reasons. First, it can be implemented in a totally machine independent form except for the operating system interface. Second, it is a useful tool that is often more convenient to use than the STAGE2 implementation. A compiler-interpreter can provide a faster development cycle for programs because it is more interactive and reduces the time between writing and testing.

The compiler-interpreter is organized as shown in figure 6. The compiler is designed along the same lines as the nucleus macros for STAGE2 described in chapter 4. It is also designed such that few modifications would be



Compiler-Interpreter Organization

Figure 6

required to generate machine code instead of MILL code.

The parser recognizes statements by top-down recursive descent. When the parser calls for the next symbol the procedure GETSYM, shown in figure 7, is used. GETSYM skips any comments and interfaces to the lexical scanner, LEX. The lexical scanner accepts the next symbol in the input file and returns appropriate information about the symbol. If the symbol is a name it is looked up in the symbol table, and if it is a single character token, such as an operator or separator, other useful attributes are returned. The intent of this approach is to provide the parser with information to determine what to do with the symbol directly and efficiently.

```
  DCL KIND;        * Kind code of current symbol
  DCL CH,NEXTCH,LEXEME(4);
  DCL STAR=('*');  * Beginning of comment symbol
*----------------
  PROC GETSYM;     * Get next non-comment symbol
    KIND=LEX(NEXTCH,LEXEME);
    CH=LEXEME(4);  * First character of symbol
    DO WHILE CH EQ STAR;
      CALL SKIP(NEXTCH);
      KIND=LEX(NEXTCH,LEXEME);
      CH=LEXEME(4);
      ENDDO
    RETURN
ENDPROC
```

The GETSYM Procedure

Figure 7

The compiler is implemented in four modules: MAIN, SYNTAX, SYNDCL, and EXPR. MAIN is the main controlling module of the compiler-interpreter and parses each source module by calls on SYNTAX. SYNTAX parses each statement and calls on EXPR for parsing expressions, and on SYNDCL for declaration statements. The SEMANT module is used by the parsing procedures to generate MILL code.

The interpreter is contained in the INTERP module and is implemented as a CASE statement. Each case is a short section of code that implements the effect of the corresponding MILL instruction. The interpreter operates by fetching the next MILL instruction at a location in the CODE vector indexed by PC, the program counter for the program being interpreted. In effect, the interpreter simulates the abstract MILL machine.

In the present organization of the compiler-interpreter, the syntax, code generation, CODE vector, and interpreter are contained in a single large program. The result of this organization is that only one source module can be interpreted at a time. A more generals organization would generate MILL code to an output file along with a compressed version of the symbol table. Separately compiled modules of this kind would be combined and executed by a separate pseudo-loader containing the

interpreter.

Lexical Scanner – LEX

The lexical scanner is used to obtain the next token from the input stream. It also takes care of reading lines from the input file and producing a listing file. LEX is called by:

```
KIND=LEX(NEXTCH,LEXEME);
```

where KIND is a code representing the kind of token that has been read, NEXCH is the next character following the token, and LEXEME is a vector of five words that contains detailed information about the token. The KIND codes returned by LEX are:

KIND	
1	Keyword
2	Assignment Operator
3	Relational Operator
4	Additive Operator
5	Multiplicative Operator
6	Integer Constant
7	String in Quotes
8	Separator
10	Local Symbol
11	Global Entry Symbol
12	External Symbol
128	Undefined Symbol

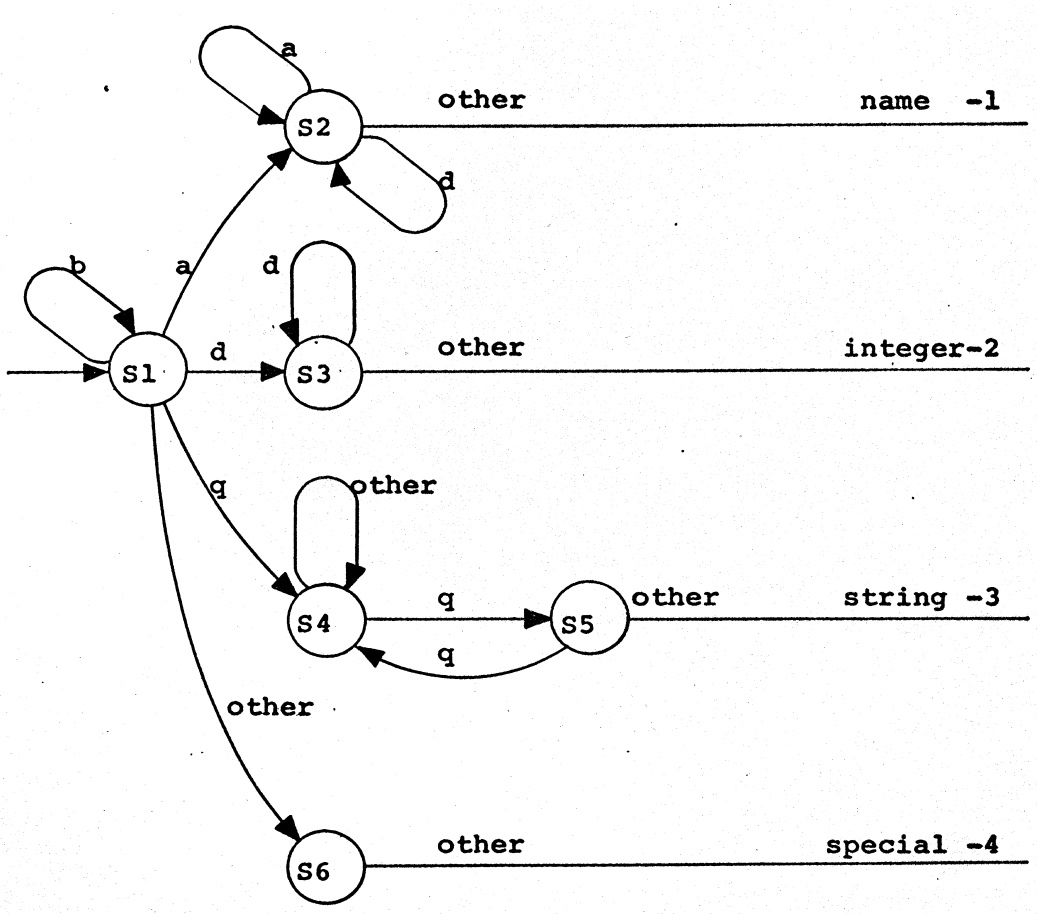
The five words of LEXEME contain details about the token as follows:

```
LEXEME(0) = Low byte of VAL (OP code if operator)
LEXEME(1) = Index of symbol in TABLE
LEXEME(2) = VAL word
LEXEME(3) = TAG word
LEXEME(4) = First character of token
```

The lexical scanner is implemented as a finite state automaton. Figure 8 shows the state diagram consisting of six states, S1 through S6, and four exits, or terminal states, designated by -1 through -4. State S1 is the starting state. This type of lexical scanner is both compact in its data structure and fast in execution.

The lexical scanner recognizes names, integers, strings enclosed in apostrophes, and special single character tokens. A name consists of a letter followed by any number of letters of digits. An integer consists of any number of digits. A string is any sequence of characters enclosed by apostrophes used as quotation marks. An apostrophe may be included in a string by writing two consecutive apostrophes. The special character tokens are the single character operators +, -, *, /, and =, and the punctuation marks (,), comma, period, and semicolon.

In the state diagram, “b” represents a blank of any other character, such as end-of-line, that is to be ignored



State Diagram for Lexical Scanner

Figure 8

in state S1. The letter “a” represents all alphabetic characters, “d” represents decimal digits, and the letter “q” represents a quotation mark. Each path labeled “other” in the diagram denotes the path taken when the current character has no other specific path to follow.

The lexical scanner is implemented in Small with a state matrix, as shown in figure 9.

	character class code				
	1	2	3	4	5
	b	a	d	q	o
state					
S1	1	2	3	4	6
S2	-1	2	2	-1	-1
S3	-2	-2	3	-2	-2
S4	4	4	4	5	4
S5	-3	-3	-3	4	-3
S6	-4	-4	-4	-4	-4

b – blank
a – alphabetic
d – digit
q – quote mark
o – other

Lexical Scanner State Matrix

Figure 9

Each row of the matrix represents the current state, and the column is the class of the current character. The positive numbers within the matrix represent the next state, and the negative numbers represent terminating states. The column number, or class code, is obtained by a translation table indexed by the character code of the current character.

Since Small does not support two-dimensional arrays in the present version, the next state matrix is implemented as a column of row vectors. An auxiliary vector is used to obtain the row offset, i.e. the index of the first element of the row relative to the origin of the matrix, from the row number. The row offset plus the column number is the index of the element corresponding to (row,column).

```
DCL S=30;          * S(0) contains array size
  DCL S1=( 1, 2, 3, 4, 6);        * offset 0
  DCL S2=(-1, 2, 2,-1,-1);        * offset 5
  DCL S3=(-2,-2, 3,-2,-2);        * offset 10
  DCL S4=( 4, 4, 4, 5, 4);        * offset 15
  DCL S5=(-3,-3,-3, 4,-3);        * offset 20
  DCL S6=(-4,-4,-4,-4,-4);        * offset 25
DCL SROW=( 6, 0, 5,10,15,20,25);  * Aux vector

STATE=S(SROW(ROW)+COL);   * Next state of (ROW,COL)
```

The auxiliary vector provides a fast method of two-dimensional array access since indexing is fast relative to multiplication by the row dimension on may computers.

Token Evaluation

Lexical tokens are evaluated by the lexical scanner in order to provide useful information about the current token to the parser. For example, the parser needs to know whether the current token is an operator or operand, a

keyword of the language, or a punctuation mark. If the token is an operator its precedence will be needed, and the KIND code returned by LEX provides this information. If the token is an integer its value will be useful, and if it is a name its value and attributes from the symbol table will be needed to determine what to do with it.

If the token being scanned is an integer its value is accumulated as it is scanned by:

```
N=10*N+CH-'0';
```

where CH is the current character code and '0' is the character code for zero. N is the value being accumulated and is the value of the integer when terminal state (-2) is reached.

If the current token being scanned is a name its first six characters are accumulated in a form suitable for symbol table access. When the terminal state for name recognition (-1) is reached the symbol table is accessed using the six characters as a key. The symbol table access returns the value and attributes of the symbol if it has been defined previously. If the symbol is not already in the table it is entered and marked undefined.

Symbol Table

The symbol table is implemented as an encapsulated data type contained in the SYMACC (symbol access) module. Symbols are stored in an array called TABLE that is indexed by a hash function on the first six characters of a name. Each symbol occupies four words in the table. This form of symbol table implementation is both compact and fast.

Symbol information is accessed by four procedures shown below.

```
Look up symbol:
   INDEX=LOOKS(WORDS,VAL,TAG);
Get symbol information:
   CALL GETS(INDEX,VAL,TAG)
Put symbol information:
   CALL PUTS(INDEX,VAL,TAG)
Get symbol base 40 code words:
   CALL GETWRD(INDEX,WORDS)
```

where

```
   INDEX  =  index of symbol in TABLE
          =  -1 if TABLE becomes full
   WORDS  =  array containing base 40 code
   VAL    =  symbol value from TABLE
```

TAG = tag word from TABLE

WORDS is a two element array containing the first six characters packed in a form called base 40. VAL is the value or location of the symbol, and the TAG word contains packed attributes. INDEX is the index of the symbol's place in TABLE, unless the table becomes full while attempting to insert the current symbol, in which case the value -1 is returned.

Base 40 is a technique that is based on the fact that a character set limited to 40 characters can be used to pack three characters in a 16 bit word. The base 40 character set includes 10 digits, 26 capital letters, blank, and three special character codes. Three characters are packed in a word P by the polynomial:

```
P=(C1*40+C2)*40+C3;
```

where C1, C2, and C3 are character codes in base 40. Note that multiplication by 40 can be reduced to two shifts and an add.

Each symbol occupies four words in which the first two words contain the first six characters packed in base 40 representation. The six characters are left justified and filled with trailing blanks. The third, or VAL word, contains the value or location of the symbol, and the

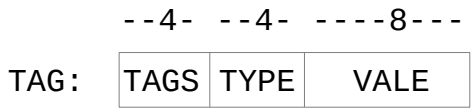
fourth, or TAG word, contains several fields of descriptive attributes.

Word	0	C1	C2	C3
	1	C4	C5	C6
	2	VAL		
	3	TAG		

Four Word Symbol Format

Figure 10

The hash function used to address the symbol table calculates a symbol number by adding the two base 40 words modulo the number of symbols the table can hold. The table size presently used is 2012 words, which provides space for 503 symbols. The size is chosen to provide a prime modulus which is just less than a power of 2. Use of a prime modulus helps randomize the symbol numbers. A symbol number is multiplied by four, by a left shift, to produce an index into TABLE. If the symbol position thus found is occupied by a symbol different from the one desired, a sequential search, with wrap around, is employed to find the desired symbol or the next available position for inserting a new symbol.



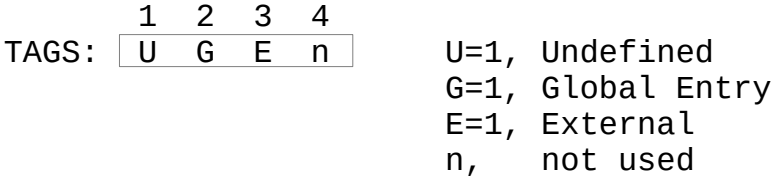
TAG Word Fields

Figure 11

The four bit TYPE field, shown in figure 11, contains a type code that is used to determine the meaning of the other fields. The TAGS field, shown in figure 12, indicates whether the symbol is local, external, a global entry, or undefined. The VALE field is an extension of the VAL field in certain cases. The TYPE codes are as follows:

TYPE

0	V	Scalar Variable
1	S	SET Parameters
2	A	Array Name
3	K	Keyword or Operator
4	P	Procedure Name (Non-recursive)
5	F	Formal Parameter Name
6	I	Integer
7	C	Character
8	R	Recursive Procedure
9	L	Label

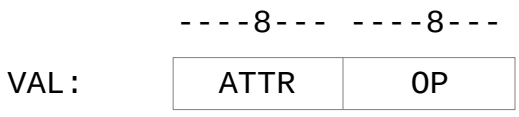


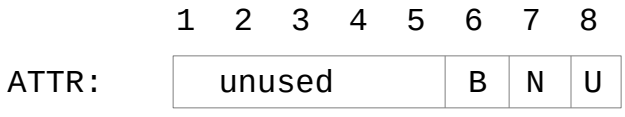
The Bits of the TAGS Field

Figure 12

When the TYPE code is 3 the symbol is a reserved word, i. e. a keyword or an operator. The KIND code is contained in the VALE field of reserved words. When TYPE = 3, and KIND = 1 (keyword), the VAL word contains a code indicating which keyword the symbol represents. By this mechanism certain keywords and their abbreviations, such as DECLARE and DCL, are indistinguishable to the parser. The keywords and their values are listed in table 3.

When the TYPE is 3 and KIND is 3, 4, or 5, the symbol is an operator. In this case the VAL word contains ATTR and OP fields, where OP indicates which operator, and ATTR indicates whether the operator is unary or binary, and if binary whether commutative or non-commutative.





B = 1, Binary Operator
N = 1, Non-Commutative Operator
N = 0, Commutative Operator
U = 1, Unary Operator

VAL Word Fields for Operators

Figure 13

Table 3

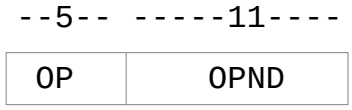
Keywords

VAL	Symbol	VAL	Symbol
1	START	30	BEGIN
2	IF	31	END
3	DO	32	THEN
4	REPEAT	33	ELSE
5	CASE	34	ENDIF
6	LABEL	35	WHILE
7	EXIT	36	ENDDO
8	GO	37	UNTIL
9	SET	38	OF
10	CALL	39	ENDCASE
11	RECURSIVE	40	TO
11	REC	41	ENDPROC
12	PROCEDURE		
12	PROC		
13	RETURN		
14	ENTRY		
14	ENT		
15	EXTERNAL		
15	EXT		
16	DECLARE		
16	DCL		
17	MESSAGE		
17	MSG		
18	STOP		

The symbol table is initialized with the reserved words, keywords, and operators, and their VAL and TAG words, when the compiler-interpreter is executed.

Code Representation

The SEMANT module contains the semantic procedures used to build the CODE vector. Each MILL instruction to be passed to the interpreter is placed in the CODE vector in the form shown in figure 14.



MILL Instruction Format in CODE Vector

Figure 14

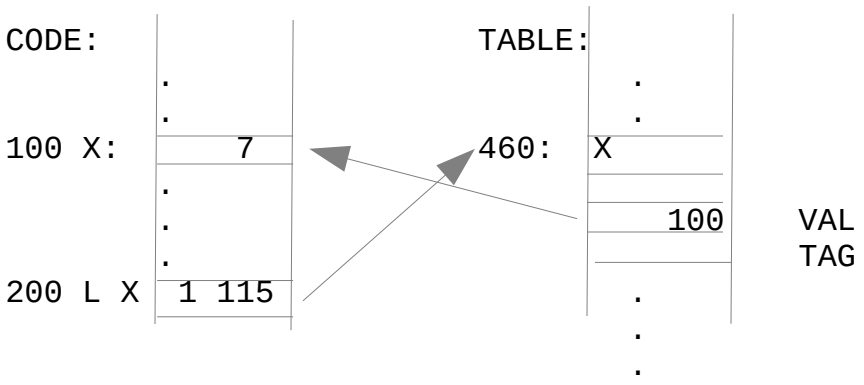
OP is an operation code as shown in table 4, and OPND is a symbol number used to address the symbol table. Symbol number zero is not used so that OPND=0 can indicate an extended set of OP codes with no address. OPND is multiplied by four to obtain an index into TABLE. The symbol at this position is the operand of the MILL instruction. The VAL word of this symbol is the value of

Table 4

MILL Interpreter Operation Codes

	Single Address			No Address	
1.	L	opnd	1.	L	*.AC
2.	L	=opnd	2.	-	
3.	ST	opnd	3.	NOT	
4.	ST	*opnd	4.	DEND	
5.	+	opnd			
6.	+	=opnd	5.	CEND	
7.	-	opnd	6.	END	
8.	*	opnd			
9.	/	opnd			
10.	AND	opnd			
11.	OR	opnd			
12.	XOR	opnd			
13.	MOD	opnd			
14.	SHR	opnd			
15.	SHL	opnd			
16.	J	opnd			
17.	JEQ	opnd			
18.	JNE	opnd			
19.	JLE	opnd			
20.	JLT	opnd			
21.	JGE	opnd			
22.	JGT	opnd			
23.	JX	opnd			
24.	SUBR	opnd			
25.	RSUBR	opnd			
26.	SCALL	opnd			
27.	RETN	opnd			
28.	RRETN	opnd			
29.	STRT	opnd			

the symbol, or the index of the actual storage location in the CODE vector. Figure 15 illustrates the interaction between the CODE vector and the symbol table.



Relationship Between CODE and Symbol Table

Figure 15

In the CODE vector shown the operand X is located at CODE(100), and an “L X” instruction to load X is at CODE(200). The OPND field of CODE(200) is 115, the symbol number of X, which is to be multiplied by four to index TABLE at TABLE(460). The operand OPND is obtained by:

```
OPND=CODE(PC) AND MASK11;
```

PC is the current program counter, 200; and MASK11 is 2047, a mask to isolate the low eleven bits of an instruction. Then

```
VAL=TABLE((OPND SHL 2)+2);
```

obtains the VAL word of symbol X, which is the locations of X in CODE. Now

```
AC=CODE(VAL);
```

loads AC, the interpreter's accumulator, with seven, the contents of X.

Chapter 7
Conclusion

In this chapter several issues are discussed which did not fit elsewhere, or which point to future directions.

Word Size vs. Portability

The version of Small described here makes no assumptions about word size except that a word must be large enough to hold an address. In practice this means that the word size must be at least 16 bits. The intention is that word size is an implementation dependent issue, and the choice of word size should be based on the capabilities of the target hardware and the programs to be run on it. The language does not adopt any particular word size as a standard so that algorithms expressed in Small can be portable across machines with different word sizes.

This view of the word size issue originated when word sizes of 18 or 20 bits, or other sizes, were considered viable design alternatives. Programs written in Small intended to have the widest portability should have their algorithms and data structures constrained to use the 16

bit minimum size. If a program is written in such a way that a larger word size is required its portability would potentially be limited, except for the fact that it is possible to implement a version of Small with a different word size by writing a handful of support routines and code generation macros to use them.

Modern hardware designs, particularly microprocessors, have almost universally adopted the eight bit byte as the unit of storage. This fact taken to its logical conclusion would suggest that a language designed for portability across such designs should also adopt the byte as the universal unit of storage. Then word sizes would be implemented and used as multiples of bytes, and portability would be assured by use of the universal storage unit.

However, the present version of Small adheres to its original design goals and does not define a standard word size apart from the 16 bit minimum. This does not prevent another version of Small from adopting the byte as the storage unit standard. One of the purposes of Small is to be the basis of a family of languages to explore such issues. A version called SmallB had been developed that adds BYTE and WORD data types, defined as 8 and 16 bits respectively. This version maps very efficiently onto most of the currently popular microprocessors and many

minicomputers.

Prime 300 Implementation

The first implementation of Small used a Prime 300 minicomputer and a reasonably well optimized version of STAGE2 in Prime assembly language. The Prime 300 has a single 16 bit accumulator called the A register, and two 16 bit registers called B and X which can be used as index registers. The Prime 300 is upward compatible with Honeywell 316 and 516 minicomputers, and can operate in any of several compatibility modes which can be set by privileged instructions.

The Prime 300 implementation of Small was quite straightforward since all of the MILL instructions mapped to one or two machine instructions. Several programs were written in Small and run on the Prime which thoroughly tested the features of the language and the details of the implementation. The Prime implementation did not include the CASE statement of optional recursion which were added later. This implementation is presently inactive due to lack of access to the Prime computer.

Intel 8080 Implementation

An implementation of Small for the Intel 8080 microprocessor has been done under the CP/M operating system. An efficient and flexible version of SGARE2 implemented by a colleague, Richard Curtiss, is being used. This version of STAGE2 has been submitted to the CP/M User Group where it is available for a nominal reproduction charge. The 8080 implementation of Small is also being submitted to the CP/M User Group and will be available when it has been cataloged. The 8080 version of Small and STAGE2 are also compatible with Intel 8085 and Zilog Z80 microprocessors.

The 8080 microprocessor contains seven 8 bit registers named A, B, C, D, E, H, and L. Some of these can be combined into 16 bit registers called BC, DE, and HL, where these register pairs can be used as 16 bit accumulators or as limited index registers. In the implementation of Small, HL is used as a 16 bit accumulator, and DE is used as a high speed temporary storage register via the XCHG instruction. The 8080 lacks 16 bit arithmetic and logical instructions for all operations except addition. The DAD instruction is used for the MILL addition instruction, and most other operations use short support routines.

Curtiss wrote parts of the run-time support for the 8080 version, and we collaborated on the interface to the CP/M operating system. The run-time support consisted of 8080 assembly language routines to perform argument transfer; the arithmetic operations: subtract, multiply, divide, modulo, and negate; the bitwise logical operations: AND, OR, XOR, and complement; and the shift operations: SHL and SHR. The interface to CP/M is written partly in Small and partly in assembly language, and consists of READ and WRITE as described in chapter 5, plus ATTACH and CLOSE. ATTACH sets up an association between a unit number, a file name, and a file block. The file block consists of 82 words that include the file control block required by CP/M, and status flags such that the associated file will be opened when the first READ or WRITE is performed. ATTACH is called by:

```
CALL ATTACH(FUNIT,FNAME,FBLOCK)
```

where FUNIT is a unit number, FNAME contains the file name, and FBLOCK is an 82 word file block. CLOSE is called with a unit number and closes the associated file.

Curtiss has written several applications in Small in connection with his consulting business. A major application written by him is BLX, a preprocessor for Basic that permits structured programming and eliminates line

numbers. He is offering BLX as a commercial product under CP/M.

Another application which I completed recently is a text formatter called Small Runoff. Small Runoff is based on Seattle Runoff written by Joseph Felsenstein of the University of Washington Department of Genetics. Seattle Runoff is written in Pascal and is based on the Format program described in [14]. Seattle Runoff changed many of the commands in Format to make them more like the versions of Runoff usually found on Digital Equipment System-10, PDP-11, and other computers. This thesis is formatted using Small Runoff running on my personal computer. Previous versions of the thesis were processed with a compiled version of Seattle Runoff running in the same environment. In one benchmark test a text file of 355 lines took 2 minutes using Seattle Runoff, and 45 seconds using Small Runoff. The Seattle Runoff version required 15k bytes (where k=1024) plus heap and stack space. The Small Runoff version required 17k bytes plus stack space; no heap is required in Small and stack space in minimal since only return addresses are stacked. With the heap requirement included, Seattle Runoff actually used more memory than Small Runoff. The speed and space advantage of Small is due primarily to its deliberate simplicity in design and implementation.

A preliminary design for a more complete language based on Small has been done. The design includes byte, integer, real, double precision, and sets as basic types, and user-defined types based on the concept of an encapsulated set of operations. The design also extends MILL to a type independent intermediate language. This version of the language is expected to lead to a commercially practical implementation language.

Bibliography

1. Bauer, F. L. ed., Compiler Construction, An Advanced Course, Springer-Verlag, Lecture Notes in Computer Science, Vol. 21, 2nd ed., New York, 1976.
2. Bauer, F. L. ed., Software Engineering, An Advanced Course, Springer-Verlag, Lecture Notes in Computer Science, Vol. 30, New York, 1975.
3. Barrett, W. A., J. D. Couch, Compiler Construction: Theory and Practice, Science Research Associates, 1979.
4. Bohm, C., G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules," Communications of the ACM, (9/5) May 1966.
5. Britten, C. R., "Small – A Simple Mobile Algorithmic Language," NCC'78 Personal Computing Digest, Anahiem 1978, pp 251-271.
6. Brown, P. J., Macro Processors and Techniques for Portable Software, Wiley and Sons, New York, 1974.
7. Campbell-Kelly, M., An Introduction to Macros, McDonald-Elsevier, New York, 1973.
8. Coleman, S., P. C. Poole, W. M. Waite, "The Mobile Programming System, Janus," Software, Practice & Experience, (4/1) Jan 1974, pp 5-23.
9. Colman, S., Janus: A Universal Intermediate Language,

Report NSF-OCA-GC32471-J2A, 1974.

10. Dahl, O.-J., E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press, New York, 1972.
11. Griswald, R. E., J. F. Poage, I. P. Polonski, The SNOBOL4 Programming Language, 2nd ed., Prentice-Hall, New Jersey, 1971.
12. Haddon, B. K., W. M. Waite, “Experience with the Universal Intermediate Language Janus,” Software, Practice & Experience, (8/4) Jul 1978, pp 601-616.
13. Halstead, M. H., Machine-Independent Computer Programming, Spartan Books, Washington D. C., 1962.
14. Kernighan, B. W., P. J. Plauger, Software Tools, Addison-Wesley, Reading, Mass., 1986.
15. Knuth, D. E., “An Empirical Study of Fortran Programs,” Software, Practice & Experience, (1/2) Apr 1972, pp 105-134.
16. Knuth, D. E., “Structured Programming with go to Statements,” Computing Surveys, (5/3) Sep 1973, pp 261-301.
17. Lavington, S. , Early British Computers, Manchester University Press, 1980.
18. Myers, G. J., The Case Against Stack-Oriented Instruction Sets, ”Computer Architecture News, (6/3) Aug 1977, pp. 7-10.
19. Orgass, R. J., H. Schorr, W. M. Waite, M. V.

Wilkes, WISP, A Self Compiling List Processing Language, Department of Electrical Engineering, University of Colorado, Feb 1967.

20. Richards, M., ":BCPL A tool for Compiler Writing and System Programming," Spring Joint Computer Conference, 1967, pp 557-566.
21. Sammet, J. E., Programming Languages: History and Fundamentals, Prentice-Hall, New Jersey, 1969.
22. Tannenbaum, A. S., "A General Purpose Macro Processor as a Poor Man's Compiler Compiler," IEEE Transactions on Software Engineering, (2/2) Jun 1976, pp 121-125.
23. Waite, W. M., Implementing Software for Non-Numeric Applications, Prentice-Hall, New Jersey, 1973.
24. Wilkes, M. V., D. J. Wheeler, S. Gill, The Preparation of Programs for an Electronic Digital Computer, Addison-Wesley, Cambridge, Massachusetts, 1951.
25. Wilkes, M. V., "An Experiment with a Self-Compiling Compiler for a Simple List Processing Language," Annual Review in Automatic Programming, MacMillan, New York, 1964.
26. Zelkowitz, M. V., W. G. Bail, "Optimization of Structured Programs," Software, Practice & Experience, (4/1) Jan 1975, pp 51-58.

Appendix A

The EDSAC Computer

The EDSAC was among the early examples of computing machinery and had an interesting history in its own right [24]. It had a number of design characteristics that remain valid and applicable to an abstract machine design. The EDSAC was related, in it design and construction, to the EDVAC, a design based on a report known as the “First Draft,” compiled by John von Neumann in 1945. The EDVAC and EDSAC were both examples of classical von Neumann machines, and were two of the first with stored programs. The EDVAC was built at the Moore School of the University of Pennsylvania, while the EDSAC was constructed by a group under Maurice Wilkes at Cambridge University, England. In May, 1949, the EDSAC became the first operational stored program computer, while the EDVAC was not completed until 1952.

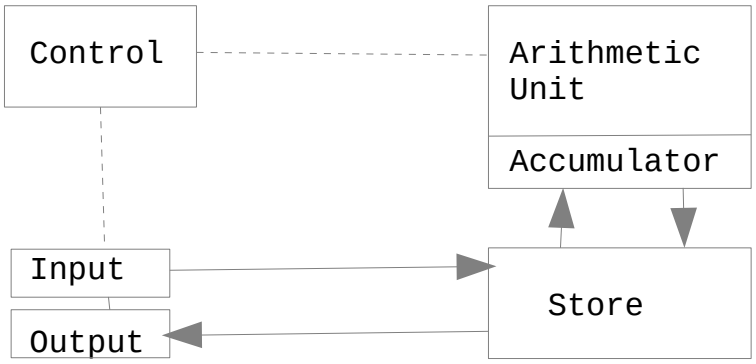
The EDSAC was a single accumulator machine with an arithmetic unit capable of binary addition, subtraction, and multiplication. Division had to be performed by a subroutine. The machine originally had a memory of 1024 words of 17 bits, known as the free store, which was later expanded by an additional 16,384 words that became known as the main store. The free store registers were used to

access the main store by indirect addressing.

The EDSAC used acoustic delay line memory implemented in mercury tanks. The 17 bit words could be combined into double words of 35 bits in which an extra bit was gained by using the timing gap that normally separated two 17 bit words. The capacity of the EDSAC is sometimes given as 512 words of 36 bits when both timing bits are counted. The arithmetic unit contained a single accumulator with a capacity of 17 bits, and a separate multiplier register. The instruction set had 18 instructions using five bit codes that corresponded to teletypewriter characters. The instruction codes used the five most significant bits of a 17 bit word, followed by 11 address bits, and the least significant bit indicated whether the operand was to be a single or double length word.

Input and output were accomplished through a Creed teleprinter and a five level paper tape reader, an arrangement similar to teletypewriters still in use today. Programs were prepared using an off-line paper tape punch and frequently made use of previously prepared subroutines. A subroutine library was maintained in a cabinet containing a collection of documented and tested paper tapes. When a subroutine was needed is was mechanically copied to the new program tape. Jobs were submitted in a batch queue that

consisted of a horizontal wire running to the machine operator's area. Job control consisted of a ticket filled out by the programmer and attached to the program tape. The operator ran the job tapes in the order received on the wire, subject to any overriding priorities, and placed the teleprinter output with the program tape in a rack ready for the programmer. The extensive mathematical subroutine library was a significant contribution to the EDSAC laboratory and was published in 1951 [24]. The EDSAC was the center of an active computing service throughout the fifties, and was followed by an EDSAC II that became operational in 1957. The EDSAC II used a microprogrammed control store and ferrite core memory.



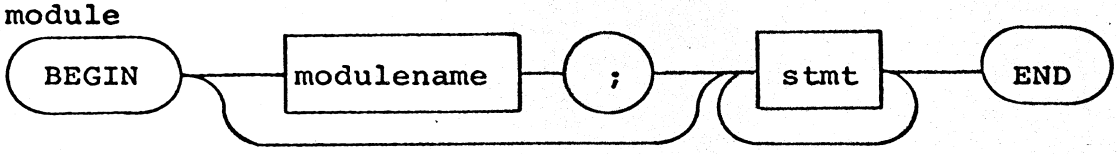
Schematic Diagram of the EDSAC Computer

Figure 16

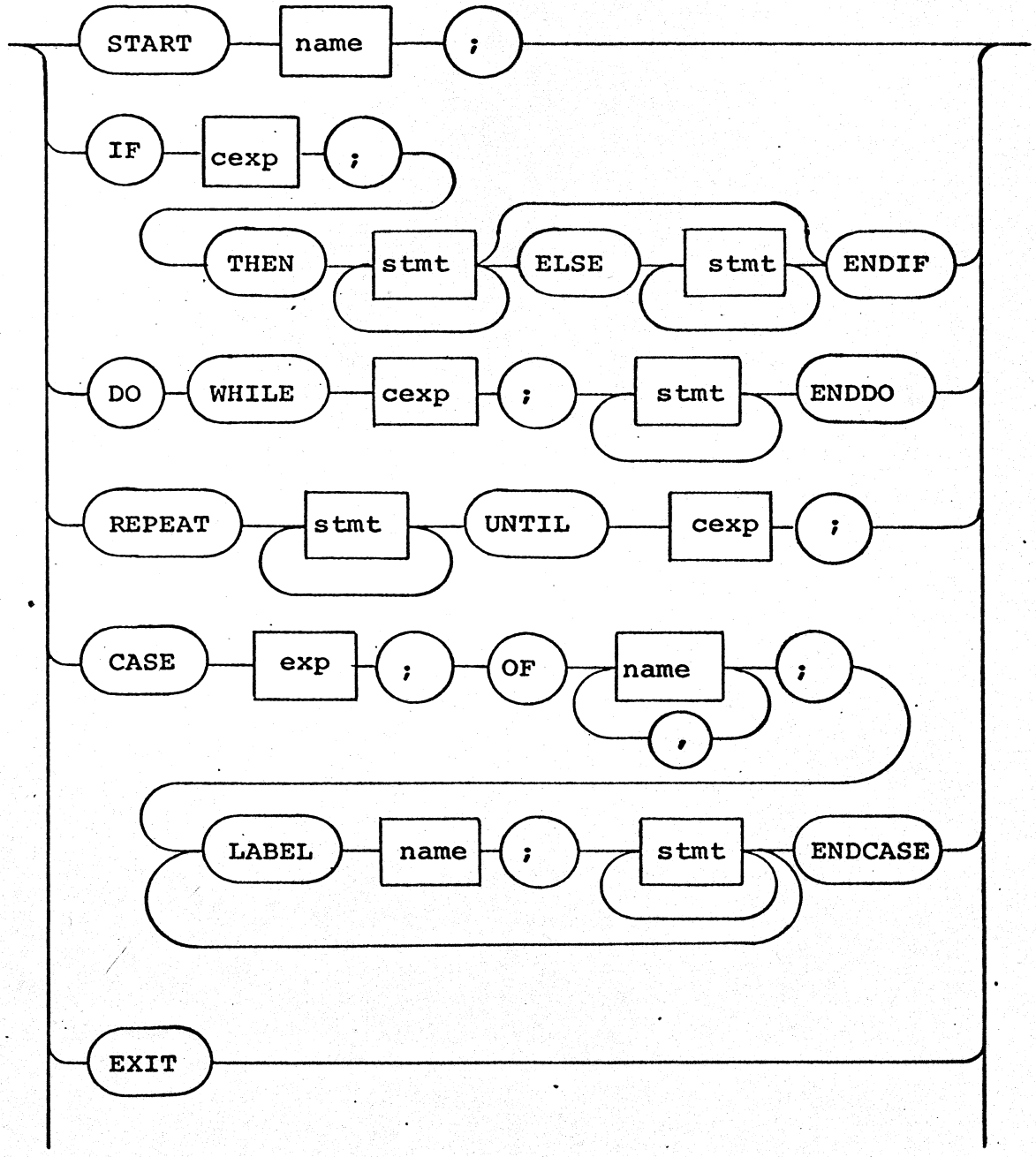
Appendix B

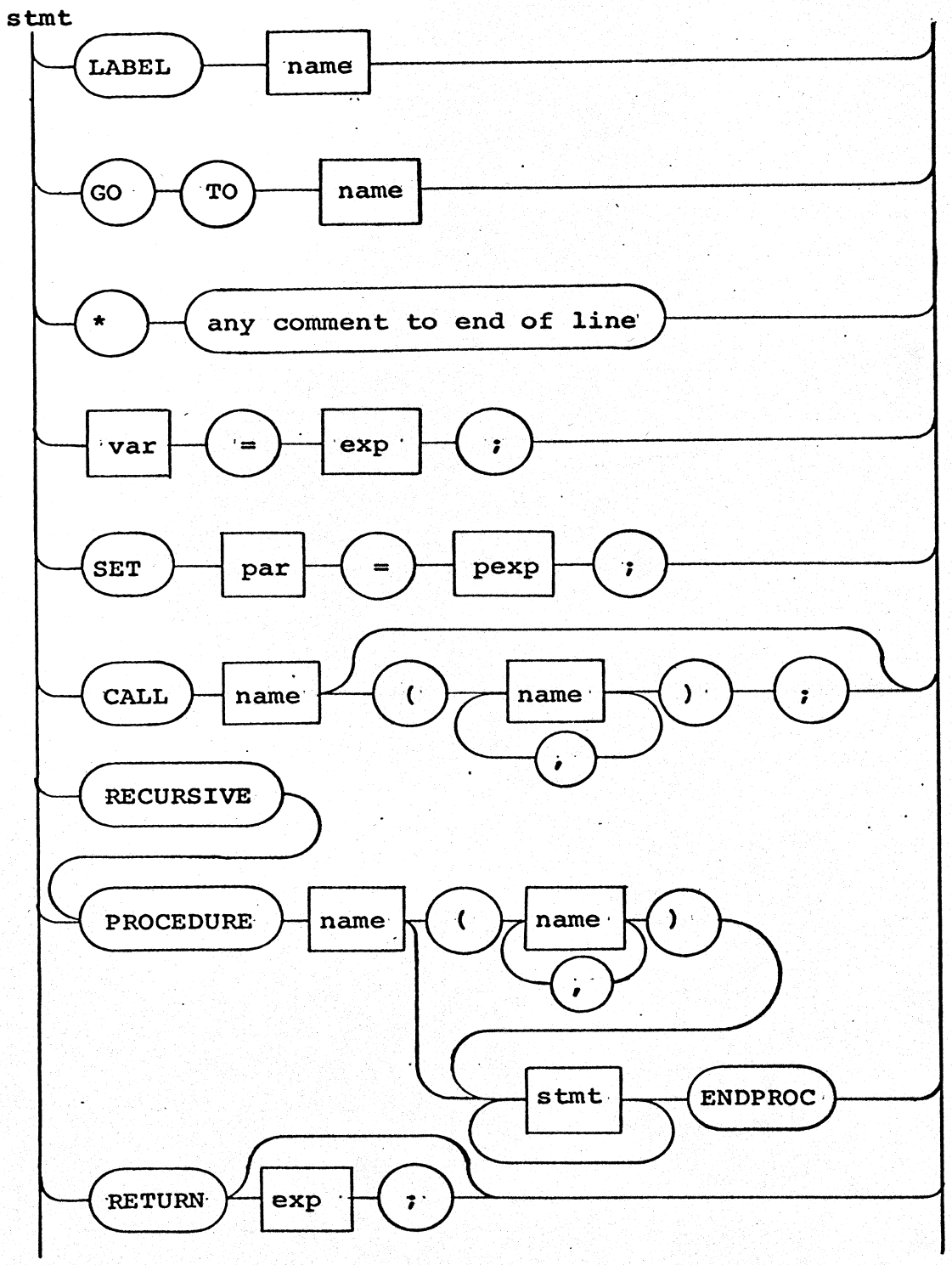
Syntax Diagrams for Small

module

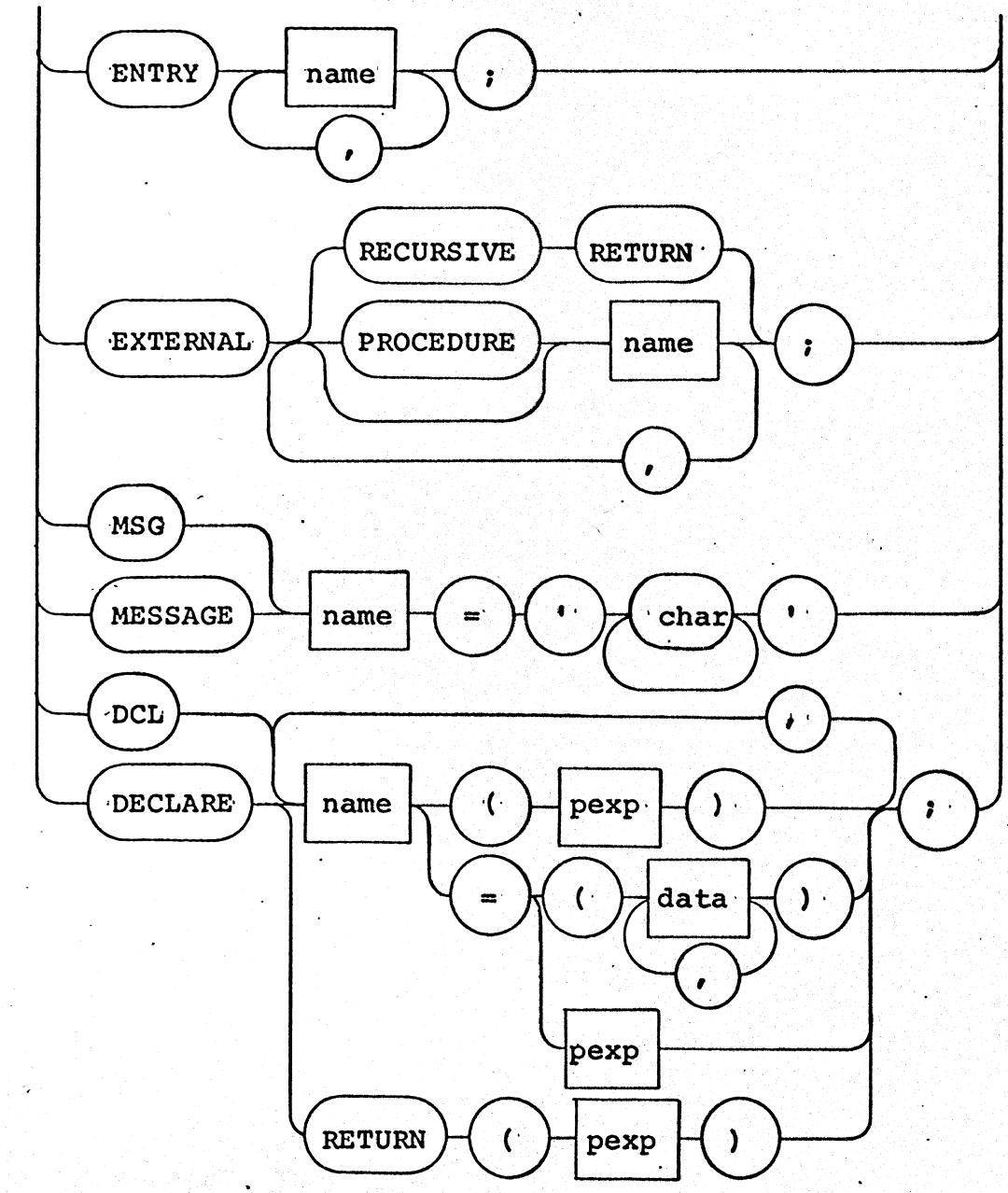

BEGIN
modulename
;
stmt
END

stmt

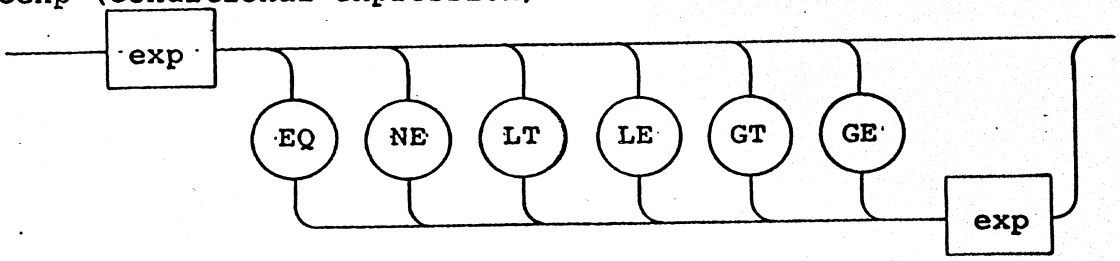

START
name
;
IF
cexp
;
THEN
stmt
ELSE
stmt
ENDIF
DO
WHILE
cexp
;
stmt
ENDDO
REPEAT
stmt
UNTIL
cexp
;
CASE
exp
;
OF
name
,
;
LABEL
name
;
stmt
ENDCASE
EXIT


stmt
LABEL
name
GO
TO
name
*
any comment to end of line
var
=
exp
;
SET
par
=
pexp
;
CALL
name
(
name
,
)
;
RECURSIVE
PROCEDURE
name
(
name
,
)
stmt
ENDPROC
RETURN
exp
;

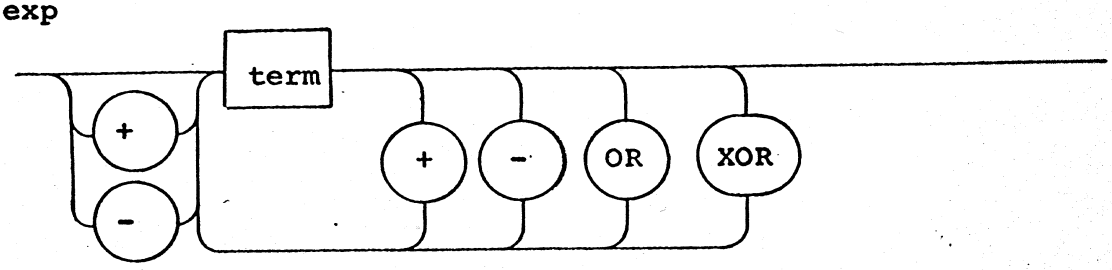
stmt


ENTRY
name
;
,
RECURSIVE
RETURN
EXTERNAL
PROCEDURE
name
;
,
MSG
MESSAGE
name
=
'
char
'
DCL
,
DECLARE
name
(
pexp
)
;
=
(
data
)
,
pexp
RETURN
(
pexp
)

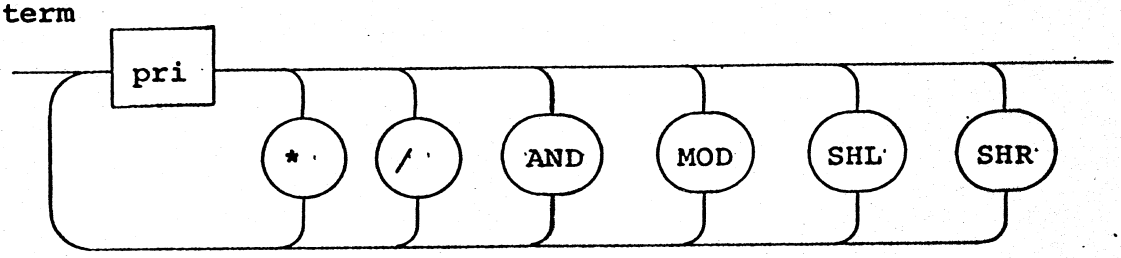
cexp (conditional expression)



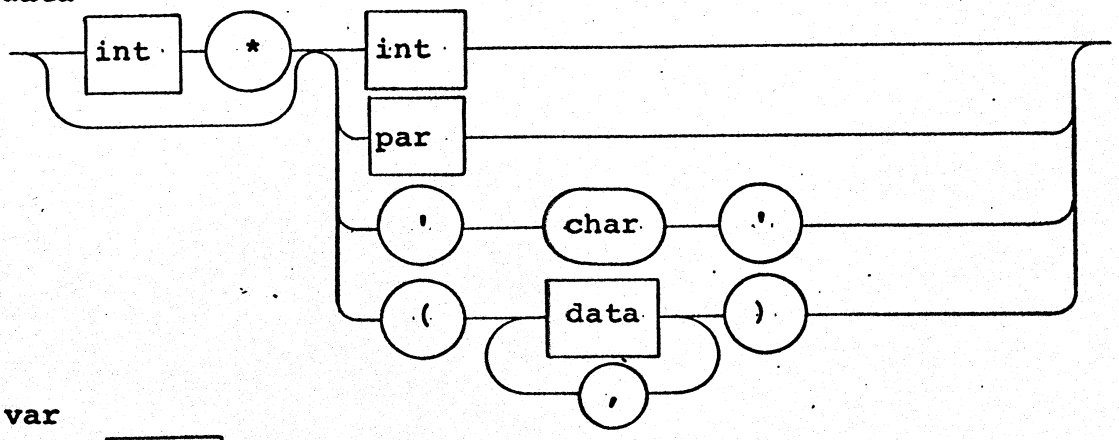
exp



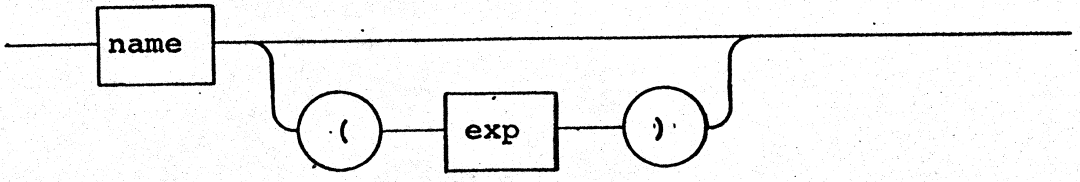
term



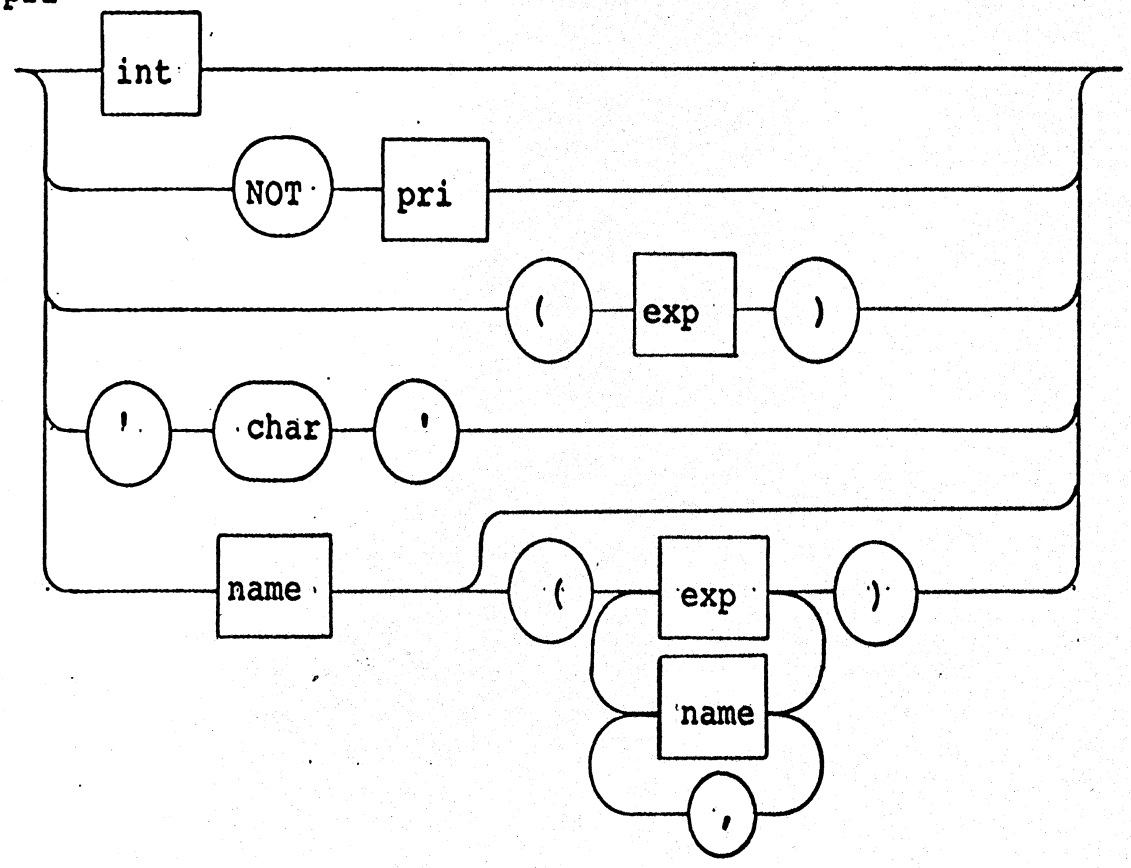
data



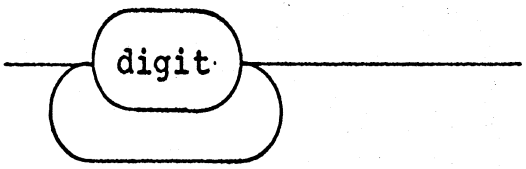
var



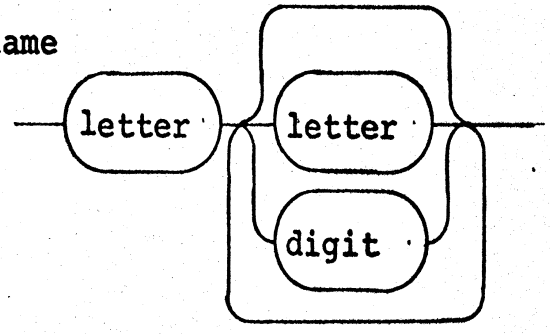
pri



int



name



Appendix C

The STAGE2 Nucleus Macros for Small

```
#$#$0 (+-*/)   FLAG LINE FOR SMALL MACRO COMPILER -- OCT 10, 1981
$#                          ACCEPT AN INOUT LINE (C) 1981 BY CR BRITTEN
*LN$26 .SET *LN=$21+1#        BUMP UP LINE NUMBER
$21$26 *LEV$36 $31$36#        RECALL LEVEL NUMBER
$F14#                         PRINT LINE NUMBER LEVEL AND 72 CHAR LINE
2222 33 11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
.INT $10#                     INTERMEDIATE LANGUAGE OUTPUT
.TLB $10#                     TRIM LEADING BLANKS
*TRIM$96 .STM $91#            PARSE TRIMMED LINE AS STATEMENT
# 1 --------------- STATEMENT PARSERS               ----------------
.STM $BEGIN $ $#            BEGIN MODULE AND INITIALIZE
.BGN#                         PROCESS BEGIN STATMENENT
.STM $30#                     NEXT STATEMENT
# 2
.STM $BEGIN $;$#            BEGIN MODULE WITH SEMICOLON
.BGN#                         PROCESS BEGIN STATMENENT
.STM $30#                     NEXT STATEMENT
# 3
.STM $BEGIN$#               BEGIN STATEMENT WITHOUT MODULE NAME
.BGN#                         PROCESS BEGIN STATMENENT
.STM $20#                     NEXT STATEMENT
# 4
.STM $START $ $#            START EXECUTION AT LABEL
.STRT $20#                    PROCESS START LABEL
.STM $30#                     NEXT STATEMENT
# 5
.STM $START $;$#            START EXECUTION AT LABEL WITH SEMICOLON
.STRT $20#                    PROCESS START LABEL
.STM $30#                     NEXT STATEMENT
# 6
.STM $STOP$#              STOP EXECUTION, CLOSE FILES, GO TO OPSYS
. CALL EXIT#
# 7
.STM $=$;$#               ASSIGNMENT STATEMENT
.INIT#                      INITIALIZE ARITHMETIC EXPRESSION ANALYZER
.VAR $10#                   VARIABLE ON LEFT HAND SIDE
.UEX $20#                   COMPILE UNARY ARITHMETIC EXPRESSION
.QUE *POS,.BNST#            BINARY NON-COMMUTATIVE STORE
*POS$96 .GEN $91#           GENERATE CODE FROM POSTFIX
.STM $30#                   NEXT STATEMENT
# 8
.STM $IF $;$#             IF CONDITIONAL
.CEXI $20#                  CONDITIONAL EXPRESSION
*POS$96 .GEN $91#           GENERATE CODE FROM POSTFIX
*LEV$76 .SET *LEV=$71+1#      INC LEVEL NUMBER
.STM $30#                   NEXT STATEMENT
# 9
.STM $THEN $#             THEN CLAUSE
*JUMP$96 . $91 LJ$01#       RECALL CONDITIONAL JUMP
.STK *TARG,$01#             STACK THE JUMP TARGET
.STM $20#                   NEXT STATEMENT
# 10
.STM $ELSE $#             ELSE CLAUSE
. J LJ$01#                  JUMP AROUND FOR TRUE CONDITION
*TARG$96 .POPT $91#         TARGET FOR FALSE CONDITION
.STK *TARG,$01#                STACK THE JUMP TARGET
.STM $20#                   NEXT STATEMENT
# 11
.STM $ENDIF$#             END OF THEN OR ELSE CLAUSE
*TARG$96 .POPT $91#         TARGET LABEL
*LEV$76 .SET *LEV=$71-1#    DEC LEVEL NUMBER
.STM $20#                   NEXT STATEMENT
# 12
.STM $EXIT$#              EXIT FROM THEN, ELSE, WHILE OR REPEAT
*TARG$96 .EXIT $91#         RECALL EXIT LABEL
```

```
.STM $20#                     NEXT STATEMENT
# 13
.STM $DO WHILE $;$#         DO-WHILE CONDITIONAL
.LABEL LJ$01#                 LOOP TARGET
.STK *LOOP,$01#               STACK LOOP TARGET
.CEXI $20#                    CONDITIONAL EXPRESSION
*POS$96 .GEN $91#             GENERATE CODE FROM POSTFIX
*JUMP$96 . $91 LJ$02#         COND JUMP TO EXIT TARGET
.STK *TARG,$02#               STACK EXIT TARGET
*LEV$76 .SET *LEV=$71+1#      INC LEVEL NUMBER
.STM $30#                     NEXT STATEMENT
# 14
.STM $ENDDO$#               END DO-WHILE
*LOOP$96 .POPL J,$91#         RECALL LOOP TARGET AND JUMP
*TARG$96 .POPT $91#           RECALL EXIT TARGET LABEL
*LEV$76 .SET *LEV=$71-1#      DEC LEVEL NUMBER
.STM $20#                     NEXT STATEMENT
# 15
.STM $REPEAT $#             REPEAT-UNTIL OPENING STATEMENT
.LABEL LJ$01#                 LOOP TARGET
.STK *LOOP,$01#               STACK LOOP TARGET
.STK *TARG,$02#               STACK EXIT TARGET
*LEV$76 .SET *LEV=$71+1#      INC LEVEL NUMBER
.STM $20#                     NEXT STATEMENT
# 16
.STM $UNTIL $;$#            UNTIL CONDITIONAL
.CEXI $20#                    CONDITIONAL EXPRESSION
*POS$96 .GEN $91#             GENERATE CODE FROM POSTFIX
*LOOP$96 *JUMP$86 .POPL $81,$91# RECALL LOOP AND COND JUMP
*TARG$96 .POPT $91#           RECALL EXIT TARGET LABEL
*LEV$76 .SET *LEV=$71-1#      DEC LEVEL NUMBER
.STM $30#                     NEXT STATEMENT
# 17
.STM $CASE $;$#             CASE STATEMENT
.CEXI $20#                    CONDITIONAL EXPRESSION
*POS$96 .GEN $91#             GENERATE CODE FROM POSTFIX
*LEV$76 .SET *LEV=$71+1#      INC LEVEL NUMBER
.STM $30#                     NEXT STATEMENT
# 18
.STM $OF $;$#               CASE LABELLIST
.CASL $01,$20#                CASE LABEL LIST FILED AWAY
.STK *TARG,$01                STACK THE TARGET FOR EXIT
.STM $30#                     NEXT STATEMENT
# 19
.STM $ENDCAS$ $#            END OF CASE STATEMENT
*TARG$96 .POPT $91#           RECALL EXIT TARGET LABEL
*LEV$76 .SET *LEV=$71-1#      DEC LEVEL NUMBER
.STM $30#                     NEXT STATEMENT
# 20
.STM $DCL $;$#              DECLARATION STATEMENT
. SECT DATA#                  CRB 08/23/2007
$20$27,#                      SPLIT DCL LIST ON COMMA
.DCL $20#                     HANDLE EACH DECLARED ITEM
$F8#                          REPEAT UNTIL LIST EMPTY
.STM $30#                     NEXT STATEMENT
# 21
.STM $DECLARE $;$#          DECLARATION STATEMENT
. SECT DATA#                  CRB 08/23/2007
$20$27,#                      SPLIT DCL LIST ON COMMA
.DCL $20#                     HANDLE EACH DECLARED ITEM
$F8#                          REPEAT UNTIL LIST EMPTY
.STM $30#                     NEXT STATEMENT
# 22
.STM $MSG $=$'$'$#          MESSAGE DECLARATION
. SECT DATA#                  CRB 08/23/2007
.LABEL $20#
. CONST $45#                  MESSAGE LENGTH
```

```
$40$47#                       BREAK MESSAGE ON EACH CHARACTER
. CONST $48#                  GENERATE ONE CHARACTER PER WORD
$F8#
.STM $50#                     NEXT STATEMENT
# 23
.STM $MESSAGE $=$'$'$#      MESSAGE DECLARATION
.STM MSG $20='$40'#
.STM $50#                     NEXT STATEMENT
# 24
.STM $GO TO $ $#            GO TO STATEMENT
. J $20#                      JUMP TO LABEL
.STM $30#                     NEXT STATEMENT
# 25
.STM $GO TO $;$#            GO TO STATEMENT WITH SEMICOLON
. J $20#                      JUMP TO LABEL
.STM $30#                     NEXT STATEMENT
# 26
.STM $LABEL $ $#            LABEL STATEMENT
. SECT CODE#                  CRB 08/23/2007
.IFC $20#                     IF CASE LABEL
.LABEL $20#
.STM $30#                     NEXT STATEMENT
# 27
.STM $LABEL $;$#            LABEL STATEMENT WITH SEMICOLON
. SECT CODE#                  CRB 08/23/2007
.IFC $20#                     IF CASE LABEL
.LABEL $20#
.STM $30#                     NEXT STATEMENT
# 28
.STM $CALL $ $#             CALL STATEMENT
. CALL $20#
.STM $30#                     NEXT STATEMENT
# 29
.STM $CALL $;$#             CALL STATEMENT WITH SEMICOLON
. CALL $20#
.STM $30#                     NEXT STATEMENT
# 30
.STM $PROC$ $ $#            PROCEDURE DECLARATION
.PROC $30#                    PROCESS PROCEDURE SETUP
.STM $40#                     NEXT STATEMENT
# 31
.STM $PROC$ $;$#            PROCEDURE DECLARATION WITH SEMICOLON
.PROC $30#                    PROCESS PROCEDURE SETUP
.STM $40#                     NEXT STATEMENT
# 32
.STM $REC$ PROC$ $ $#       RECURSIVE PROCEDURE DECLARATION
.RPROC $40#                   PROCESS PROCEDURE SETUP
.STM $50#                     NEXT STATEMENT
# 33
.STM $REC$ PROC$ $;$#       REC PROCEDURE DECLARATION WITH SEMICOLON
.RPROC $40#                   PROCESS PROCEDURE SETUP
.STM $50#                     NEXT STATEMENT
# 34
.STM $RETURN $;$#           RETURN STATEMENT
.INIT#
.UEX $20#                     COMPILE EXPRESSION
*POS$96 .GEN $91#            GENERATE EXPRESSION CODE
*RECPRC$76#                   RECALL RECURSIVE PROC FLAG
*NPARS$86 *RID$96#            RECALL RETURN ID AND NUMBER OF PARAMS
. $71RETN $91,$81#
.STM $30#                     NEXT STATEMENT
# 35
.STM $RETURN$#              RETURN FROM SUBROUTINE
*RECPRC$76#                   RECALL RECURSIVE PROC FLAG
*NPARS$86 *RID$96#            RECALL RETURN ID AND NUMBER OF PARAMS
. $71RETN $91,$81#
.STM $20#                     NEXT STATEMENT
```

```
# 36
.STM $ENDPROC$#            END PROCEDURE
.ALLOC#                      ALLOCATE ANY TEMPORARIES
*PARLIST$96#                 RECALL THE PARAMETER LIST
*LEV$76 .SET *LEV=$71-1#     DEC LEVEL NUMBER
.IF $91= SKIP 3#
$91$87,#                     BREAK PARAMETER LIST ON COMMA
.LET $80=#                   AND NULL EACH ONE OUT
$F8#
.LET *RECPRC=#               NULL THE RECURSIVE PROC FLAG
.STM $20#                    NEXT STATEMENT
# 37
.STM $ENT$ $;$#            GLOBAL ENTRY POINTS FOR THIS MODULE
$30$37,#                     BREAK PARAMETER LIST ON COMMA
. ENT $30#                   ISSUE PSEUDO-OP FOR EACH ENTRY
$F8#
.STM $40#                    NEXT STATEMENT
# 38
.STM $EXT$ $;$#            GLOBAL EXTERNAL ENTRY POINTS
$30$37,#                     BREAK PARAMETER LIST ON COMMA
.EXT $30#                    PROCESS EACH EXTERNAL
$F8#
.STM $40#                    NEXT STATEMENT
# 39
.STM $SET $=$;$#           SET COMPILE TIME VALUE
.LET $20=$34#
.STM $40#                    NEXT STATEMENT
# 40
.STM $END$#                END OF MODULE
.ALLOC#                      ALLOCATE ANY TEMPORARIES
*LEV$76 .IF $71 EQ 0 SKIP 2#
 LEVEL ERROR $71$F14#
*NERRS$96 .SET *NERRS=$91+1#
*NERRS$86  $81 ERRORS DETECTED$F14#
. END#                       END ASSEMBLY LANGUAGE MODULE
$F0#                         TERMINATE STAGE2
# 41
.STM $;$#                  DISCARD STRAY SEMICOLON
.STM $20#                    NEXT STATEMENT
# 42
.STM $*$#                  COMMENT TO END OF LINE
# 43
.STM #                     ABSORB THE NULL STATEMENT
# 44
.STM $#                    NO KEYWORD RECOGNIZED
THE FOLLOWING STATEMENT NOT RECOGNIZED$F14#
$10$F14#
*NERRS$96 .SET *NERRS=$91+1#
# 45
.STM  $#                   TRIM LEADING BLANKS
.TLB $10#
*TRIM$96 .STM $91#
# 46 --------------- CONDITIONAL EXPRESSION PARSERS    ---------------
.INIT#                     INITIALIZE ARITHMETIC EXPRESSION ANALYZER
.LET *TMP=0#                 RESET TEMPORARY STORAGE COUNTER
.LET *POS=#                  NULL THE POSTFIX QUEUE
.LET *OPN=#                  NULL THE OPERAND STACK
.LET *CC=#                   CONDITION CODES NOT SET
# 47
.CEXI $#                   CONDITIONAL EXPRESSION INITIALIZE
.INIT#                       INITIALIZE THE ANALYZER
.CEX $10#                    COMPILE CONDITIONAL EXPRESSION
# 48
.CEX $#                    CONDITIONAL EXPRESSION ANALYSIS
.LET *JUMP=JEQ#              JUMP IF ACC=0 AFTER EXP
.UEX $10#
# 49
```

```
.CEX $ EQ $#               EQUAL
.LET *JUMP=JNE#
.UEX $10-($20)#
# 50
.CEX $ NE $#               NOT EQUAL
.LET *JUMP=JEQ#
.UEX $10-($20)#
# 51
.CEX $ LT $#               LESS THAN
.LET *JUMP=JGE#
.UEX $10-($20)#
# 52
.CEX $ GE $#               GREATER THAN OR EQUAL
.LET *JUMP=JLT#
.UEX $10-($20)#
# 53
.CEX $ LE $#               LESS THAN OR EQUAL
.LET *JUMP=JGT#
.UEX $10-($20)#
# 54
.CEX $ GT $#               GREATER THAN
.LET *JUMP=JLE#
.UEX $10-($20)#
# 55 --------------- UNARY ARITHMETIC OPERATION PARSERS ---------------
.UEX $#                     NO UNARY OPERATOR
.EXP, $10#
# 56
.UEX +$#                   UNARY PLUS DELETED
.EXP, $10#
# 57
.UEX -$#                   UNARY MINUS OPERATOR
.EXP.U-, $10#
# 58 --------------- EXPRESSION PARSERS                ---------------
.EXP$, $#                  ARITHMETIC EXPRESSION
.TRM, $20#
.QUE *POS,$10#
# 59
.EXP$, $+$#                BINARY COMMUTATIVE ADD
.TRM, $20#
.QUE *POS,$10#
.EXP.BC+, $30#
# 60
.EXP$, $-$#                BINARY NON-COMMUTATIVE SUBTRACT
.TRM, $20#
.QUE *POS,$10#
.EXP.BN-, $30#
# 61
.EXP$, $ OR $#             BINARY COMMUTATIVE INCLUSIVE OR
.TRM, $20#
.QUE *POS,$10#
.EXP.BCOR, $30#
# 62
.EXP$, $ XOR $#            BINARY COMMUTATIVE EXCLUSIVE XOR
.TRM, $20#
.QUE *POS,$10#
.EXP.BCXOR, $30#
# 63 --------------- TERM PARSERS                      ----------------
.TRM$, $#
.PRI $20
.QUE *POS,$10#
# 64
.TRM$, $*$#                BINARY COMMUTATIVE MULTIPLY
.PRI $20#
.QUE *POS,$10#
.TRM.BC*, $30#
# 65
.TRM$, $/$#                BINARY NON-COMMUTATIVE DIVIDE
```

```
.PRI $20#
.QUE *POS,$10#
.TRM.BN/, $30#
# 66
.TRM$, $ MOD $#            BINARY NON-COMMUTATIVE MODULUS
.PRI $20#
.QUE *POS,$10#
.TRM.BNMOD, $30#
# 67
.TRM$, $ AND $#            BINARY COMMUTATIVE AND
.PRI $20#
.QUE *POS,$10#
.TRM.BCAND, $30#
# 68
.TRM$, $ SH$ $#            BINARY NON-COMMUTATIVE SHIFT
.PRI $20#
.QUE *POS,$10#
.TRM.BNSH$30, $40#
# 69 --------------- PRIMARY PARSERS                  ---------------
.PRI $#                   PRIMARY OPERAND
$10$871234567890#          BREAK OPERAND ON DIGIT
.IF $80= SKIP 2#
.QUE *POS,$10#             QUEUE AN IDENTIFIER OPERAND
$F9#                       IMMEDIATE EXIT FROM LOOP AND MACRO
.QUE *POS,=$10#            QUEUE NUMERIC OPERAND AS IMMEDIATE
# 70
.PRI ($)#                  RECURSIVE SUBEXPRESSION
.UEX $10#
# 71
.PRI $($)#                 ARRAY REFERENCE OR FUNCTION CALL
.IF $11=P SKIP 3#            TEST FOR PROCEDURE TAG
.VAR $10($20)#               PROCESS ARRAY VARIABLE
.QUE *POS,.UA#
$F9#
.QUE *POS,($20),.UF$10#       QUEUE ARG LIST AND FUNCTION CALL
# 72
.PRI NOT $#                UNARY LOGICAL INVERSION OF EACH BIT
.PRI $10#
.QUE *POS,.UNOT#
# 73
.PRI '$'#                  LITERAL CHARACTER
.QUE *POS,=$18#                    QUEUE CHAR CODE AS IMMED OPERAND
# 74 --------------- VARIABLE PARSERS                 ---------------
.VAR $#                     SCALAR VARIABLE
.QUE *POS,=$10#
# 75
.VAR $($)#                 ARRAY ELEMENT
.EXP, =$10+($20) SHL 1#             EVALUATE ARRAY REFERENCE
# 76 --------------- STACKING AND QUEUEING            ---------------
.STK $,$#                  STACK P2 ON STACK NAMED P1
$20,$11$26 $F3#
# 77
.QUE $,$#                  QUEUE P2 ON THE QUEUE NAMED P1
$11$20,$26 $F3#
# 78
.QUE $,#                   IGNORE NULL STRING QUEUE
# 79
.POPT $,$#                 POP TARGET LABEL
.LABEL LJ$10#                GENERATE THE LABEL
.LET *TARG=$20#              REST OF STACK
# 80
.POPL $,$,$#               POP LOOP LABEL AND JUMP CONDITIONALLY
. $10 LJ$20#
.LET *LOOP=$30#
# 81
.LABT $,$#                 JUMP TO CASE EXIT LABEL ON TOP OF STACK
. J LJ$10#
```

```
# 82
.EXIT $,$,$#               EXIT TO OUTER LEVEL
. J LJ$20#
# 83 --------------- CODE GENERATORS                ----------------
.GEN $#                    GENERATE CODE FROM POSTFIX STRING
.INT .GEN $10#               PRINT OUT CALL IN INTERMEDIATE CODE
$10$27,#                     BREAK ON COMMA TO GET POSTFIX TOKEN
$20$37.#                     BREAK TOKEN ON PERIOD TO CHECK FOR OPERATORS
.IF $30= SKIP 2#
.STK *OPN,$20#               STACK OPERAND
.SKIP 3#
*OPN$96 $20 $91#             PROCESS OPERATOR
.SKIP 1#
$F8#                         TERMINATE BREAK ON PERIOD
$F8#                         LOOP ON COMMA UNTIL POSTFIX EMPTY
*OPN$96 .IF $91= SKIP 2#       CHECK FOR NULL OPND STACK
.IF $91=.AC, SKIP 1#         IGNORE STACKED ACCUM
. L $91#                     LOAD A LONE OPERAND
# 84
.BC$ .AC,$,$#              COMMUTATIVE OP ON ACCUM
. $10 $20#                   APPLY OP TO 2ND OPND
.LET *OPN=.AC,$30#           STACK .AC
# 85
.BN$ .AC,$,$#              NON-COMMUTATIVE OP ON ACCUMULATOR
*TSET$76#                    RECALL CURRENT TEMP SET NUMBER
. ST T$71Z#                  SAVE ACCUM IN TEMP
. L $20#                     LOAD 2ND OPERAND
. $10 T$71Z#                 APPLY OP TO TEMP
.LET *OPN=.AC,$30#           STACK .AC
.LET *TUSE=1#                NOTE TNZ IN USE
# 86
.BC$ $,$,$#                FIRST OPND IS NOT ACCUM
.LET *OPN=$40#               POP 2 OPNDS OFF STACK
.IF $30=.AC SKIP 2#          SKIP IF 2ND OPND IS ACCUM
.ACB $40#                    MODIFY STACK IF ACCUM IS BUSY
. L $30#                     LOAD NEW OPND INTO ACCUM
. $10 $20#                   APPLY OP TO 1ST OPND
.STK *OPN,.AC#
# 87
.BN$ $,$,$#                FIRST OPND IS NOT ACCUM
.LET *OPN=$40#               POP 2 OPNDS OFF STACK
.IF $30=.AC SKIP 2#          SKIP IF 2ND OPND IS ACCUM
.ACB $40#                    MODIFY STACK IF ACCUM IS BUSY
. L $30#                     LOAD NEW OPND INTO ACCUM
. $10 $20#                   APPLY OP TO 1ST OPND
.STK *OPN,.AC#
# 88
.UF$ $,$#                  FUNCTION CALL UNARY OP
.LET *OPN=$30#               POP ONE OPND OFF STACK
.ACB $30#                    TAKE CARE OF BUSY ACCUM
. CALL $10$20#
.STK *OPN,.AC#
# 89
.U$ .AC,$#                 UNARY OPERATOR ON ACCUM
. $10#
# 90
.U$ $,$#                   UNARY OPERATOR ON OPERAND
.LET *OPN=$30#               POP ONE OPERAND OFF STACK
.ACB $30#                    STORE ACCUM IF BUSY
. L $20#                     APPLY UNARY OP TO ACCUM
. $10#
.STK *OPN,.AC#
# 91
.UA .AC,$#                 UNARY ARRAY REFERENCE
. L *.AC#                    ACCUM CONTAINS OPERAND ADDRESS
# 92
.ACB $.AC$#                MODIFY STACK IF ACCUM BUSY
```

```
*TMP$96 .SET *TMP=$91+1#       BUMP TEMP COUNTER
*TMAX$86#
.IF $81 GT $91 SKIP 1#
.LET *TMAX=$91#
*TSET$76#                        REPLACE TEMP SET NUMBER
.LET *OPN=$10T$71Z$91$20#    REPLACE .AC REF BY TEMP REF
. ST T$71Z$91#                   STORE ACCUM IN TEMP
# 93
.ACB $#                      NO-OP IF ACCUM NOT BUSY
# 94
.BNST .AC,$,$#               NON-COMMUTATIVE STORE RESULT FROM ACCUM
. ST *$10#
.LET *OPN=$20#
# 95
.BNST $,$,$#                 NON-COMMUTATIVE STORE
. L $10#
. ST *$20#
.LET *OPN=$30#
# 96
.BNST $,.AC,$#               STORE, TARGET ADDRESS IN ACCUM
*TSET$76#                        RECALL TEMP SET NUMBER
. ST T$71Z#
. L $10#
. ST *T$71Z#
.LET *OPN=$20#
.LET *TUSE=1#                    NOTE TNZ IN USE
# 97 --------------- AUXILLIARY STATEMENT PROCESSORS  ---------------
.BGN#                        BEGIN MODULE AND INITIALIZE
.BEGIN#                          INITIALIZE MACHINE DEPENDENT CODE GEN
.LET *NERRS=NO#                  INITIALIZE ERROR COUNTER
.SET NO=0#                       TO ZERO
# 98
.STRT $#                     START DECLARATION
. ENT $10#                       DECLARE AS GLOBAL ENTRY POINT
. STRT $10#                      GENERATE START CODE IF ANY
# 99
.PROCI $#                    PROCEDURE ENTRY INITIALIZATION
.ALLOC#                          ALLOCATE ANY TEMPORARIES
.LET *NPARS=0#                   NUMBER OF PARAMETER VARIABLES
.LET *RID=$10#                   REMEMBER RETURN ID
.LET $10=P#                      RECORD TYPE PROC
*LEV$76 .SET *LEV=$71+1#     INC LEVEL NUMBER
# 100
.$PROC $#                    PROCEDURE DECLARATION
.PROCI $20#                      INITIALIZE
. SECT CODE#                     CRB 08/23/2007
.LET *RECPRC=$10#
. SUBR $20#
.IF $10= SKIP 1#
. RECUR $20#
. NPARS 0#
.LET *PARLIST=#                  NULL THE PARAMETER LIST
. PEND#                          PROC DEFINITION END
# 101
.$PROC $($)#                 PROCEDURE DECLARATION WITH PARAMETERS
. SECT CODE#                     CRB 08/23/2007
.PROCI $20#                      INITIALIZE
.LET *RECPRC=$10#
$30$37,#                         SPLIT PARAMETER LIST ON COMMA
*NPARS$96 .SET *NPARS=$91+1#   COUNT PARAMETER VARIABLES
$F8#
. SUBR $20#                      SUBROUTINE ENTRY
.IF $10= SKIP 1#
. RECUR $20#
*NPARS$96 . NPARS $91#           NUMBER OF PARAMETER VARIABLES
$30$37,#
. PAR $30#                       ISSUE PSEUDO-OP FOR EACH PARAM
```

```
.LET $30=D#                   RECORD TYPE OF EACH PARAM
$F8#
.LET *PARLIST=$30,#           SAVE THE PARAMETER LIST
. PEND#
# 102
.EXT PROC$ $#               EXTERNAL PROCEDURE REFERENCE
.LET $20=P#                   MARK PROC TYPE
. EXT $20#
# 103
.EXT $#                     EXTERNAL REFERENCE
. EXT $10#
# 104
.EXT REC$ RETURN#           EXTERNAL RECURSIVE RETURN STACK
. EXT SKSIZE#
. EXT SK#
# 105
. CALL $($)#                PROCEDURE CALL WITH ARGS
.LET *NARGS=0#                INIT ARG COUNT TO ZERO
.LET *NCALL=$01#              SET UNIQUE NUMBER FOR THIS CALL
$20$27,#                      SCAN THROUGH THE ARG LIST
.EVAL $20#                    EVALUATE EACH ACTUAL PARAMETER
$F8#
. SCALL $10#                  CALL THE PROCEDURE
*NARGS$96 . NARGS $91#        NUMBER OF ACTUAL PARAMETERS
.LET *NARGS=0#                COUNT THEM AGAIN
$20$27,#                      SCAN THE ARG LIST AGAIN
. ARG $20#                    PUT AN ARG FOR EACH ACTUAL PARAM
$F8#
. CEND#                       END OF CALLING SEQUENCE
# 106
. CALL $#                   PROCEDURE CALL WITH NO ARGS
.LET *NARGS=0#
. SCALL $10#
. NARGS 0#                    NUMBER OF ACTUAL PARAMS IS ZERO
. CEND#
# 107
.EVAL $#                    EVALUATE AN ACTUAL PARAMETER
*NARGS$96 .SET *NARGS=$91+1#   COUNT IT
.IF $11= SKIP 1#              SEE IF THE ACTUAL IS A FORMAL
. ARGT $10#                   IF SO ARRANGE A TRANSFER
# 108
.DCL  $#                    TRIM LEADING BLANKS
.DCL $10#
# 109A                        Added TLB macro CRB 09/08/2007
.DCL $($)#                  DECLARE AN ARRAY
.LABEL $10#                   NAME OF ARRAY
$24+1$26#                     RESERVE N+1 WORDS
. SPACE $24#                  SPACE FOR ARRAY, COMPILE TIME EXP
# 109
.DCL $=$#                   DECLARE INITIALIZED VARIABLE
.LABEL $10#                   NAME OF VARIABLE
. CONST $24#                  DATA WORD CONSTANT, COMPILE TIME EXP
# 110
.DCL $#                     DECLARE SCALAR VARIABLE
.LABEL $10#                   NAME OF VARIABLE
. SPACE 1#                    SPACE FOR VARIABLE
# 111
.DCL $=($)#                 DECLARE INITIALIZED ARRAY LIST
.LABEL $10#
$20$27,#
.DATA $20#
$F8#
# 112
.DCL REC$ RETURN($)#        DECLARE RECURSIVE RETURN SPACE
. ENT SKSIZE#
. ENT SK#
. LABEL SKSIZE#
```

```
. CONST $24#
. LABEL SK#
. CONST 1#
$20-1$36 . SPACE $34#
# 113
.LABEL $#                  TRIM LEADING BLANKS FROM LABEL
. LABEL $10#
# 114
.DATA  $#                  TRIM LEADING BLANKS
.DATA $10#
# 115A                     Added TLB macro CRB 09/08/2007
.DATA $#                   DATA CONSTANT
$10$17,#
.CNST $10#
$F8#
# 115
.DATA '$'#                 CONVERT CHAR TO CONST
. CONST $18#
# 116
.DATA ($)#                 DATA SUBLIST
.DATA $10#
# 117
.CNST $#                   CONSTANT
. CONST $14#                 COMPILE TIME EXPRESSION
# 118
.CNST $*$#                 RECURSION ON REPEATED CONSTANT
$10$F7#
.DATA $20#
$F8#
# 119
.IFC $#
*CASN$76 *TARG$86 *CASL$71$96#
.IF $91= SKIP 4#
$91$27,#
.IF $10-=$20 SKIP 1#
.LABT $81#
$F8#
# 120
.CASL $,$#                 PROCESS CASE LABEL LIST
.LET *CASN=$10#
.LET *CASL$10=$20#
. JX $10#                    INDEXED JUMP THROUGH TABLE
$20$27,#
. JC $20#                    CASE TABLE ENTRY POINTS TO CASE LABEL
$F8#
# 121
. L $,#                    DANGLING COMMA ON LONE LOAD
. L $10#
# 122
. $ *=$#                   INDIRECT AND LITERAL CANCEL EACH OTHER
. $10 $20#
# 123
.ALLOC#                    ALLOCATE TEMPORARY STORAGE
*TSET$76#                    RECALL THE TEMP SET NUMBER
*TUSE$66#
.IF $61= SKIP 4#             ALLOCATE TNZ IF USED CRB 09/06/2007
. SECT DATA#                 CRB 09/06/2007
.LABEL T$71Z#
. SPACE 2#                   CRB temp hack 07/03/2013
. SECT CODE#                 CRB 09/06/2007
.LET *TMP=1#
*TMAX$96#                    RECALL MAX NUMBER OF TEMPS
.IF $91 EQ 0 SKIP 8#         SKIP IF MORE CRB 09/06/2007
. SECT DATA#                 CRB 09/06/2007
$91$F7#
*TMP$86#
.LABEL T$71Z$81#
```

```
. SPACE 1#                    SPACE FOR ONE TEMPORARY
.SET *TMP=$81+1#
$F8#
. SECT CODE#                  CRB 09/06/2007
.LET *TMP=0#
.LET *TUSE=#
.LET *TMAX=0#
.SET *TSET=$71+1#
# 124
.TLB            $#          TRIM LEADING BLANKS
.TLB $10#
#
.TLB      $#
.TLB $10#
#
.TLB     $#
.TLB $10#
#
.TLB    $#
.TLB $10#
#
.TLB   $#
.TLB $10#
#
.TLB  $#
.TLB $10#
#
.TLB $#
.LET *TRIM=$10#
# ---------------- STAGE2 SYSTEM MACROS                ----------------
.IF $=$ SKIP $#
$F50#
#
.IF $-=$ SKIP $#
$F51#
#
.IF $ EQ $ SKIP $#
$F60#
#
.IF $ GT $ SKIP $#
$F6+#
#
.SKIP $#
$F4#
#
.LET $=$#
$F3#
#
.SET $=$#
$24$26 $F3#
#
. LABEL $#                  PASS ON A LABEL WITHOUT VALUE
.. LABEL $10#
#
. $#                        INTERCEPT PSEUDO MACHINE CODE
.. $10#
#
..$#                        OUTPUT ASSEMBLY LANGUAGE STMT
$10$F13#                      INSERT MACHINE DEPENDENT MACROS NEXT
#
.BEGIN#                     DUMMY INITIALIZATION
 BEGIN$F13#                   moved to mill2nasm CRB 08/22/2007
#
. SECT $#                   Added pseudo-op CRB 08/23/2007
 SECT $10$F13#
#
.INT $#                     INTERMEDIATE LANGUAGE OUTPUT
```

```
*$10$F13#
#
. $ =$#                     try to fix dummy array ref CRB 07/12/2013
.. $10 $21 =$20#
#
. $ $#                      INTERCEPT ABSTRACT MACHINE INST
.. $10 $21 $20#
## --------------- END OF MACROS                    ---------------
```

Appendix D

Example Support Routines in Small

```
1       * CAT2 -- CONCATENATE THE MESSAGES MSG1 AND MSG2
2       *
3           BEGIN CAT2;
4           ENT CAT2;
5           DCL I,J;
6       *---------------
7           PROC CAT2(MSG1,MSG2);
8    1        I=MSG1+1;
9    1        J=1;
10   1        DO WHILE J LE MSG2;     * COPY MSG2 INTO AND FOLLOWING MSG1
11   2          MSG1(I)=MSG2(J);
12   2          I=I+1;
13   2          J=J+1;
14   2          ENDDO
15   1        MSG1=MSG1+MSG2;         * SET LENGTH OF RESULT MESSAGE
16   1        RETURN
17   1        ENDPROC
18   0      END
 NO ERRORS DETECTED


1       * IFORM -- CONVERT INTEGER TO DECIMAL CHARACTER STRING AS IN I FORMAT
2       BEGIN IFORM;                  * CRB OCT 31, 1981
3            ENT IFORM;
4            DCL N,Q;
5            DCL PTR;                 * TEMPORARY STORAGE FOR CONVERSION
6            DCL STAR=('*');
7            DCL MINUS=('-');
8       *---------------
9       PROC IFORM(NUM,BUFF);
10   1       N=NUM;                   * GET NUMBER TO BE CONVERTED
11   1       PTR=BUFF;                * POINTER INTO USERS BUFFER
12   1       IF N LT 0; THEN N=-N; ENDIF
13   1       REPEAT                   * REPEAT UNTIL N IS CONVERTED
14   2         IF PTR LE 0; THEN GO TO TOOBIG ENDIF
15   2         Q=N/10;                * GET NEXT QUOTIENT
16   2         BUFF(PTR)=N-10*Q+'0';  * CONVERT REMAINDER TO CHAR
17   2         N=Q;
18   2         PTR=PTR-1;
19   2         UNTIL N EQ 0;
20   1       IF NUM LT 0; THEN
21   2         IF PTR LE 0; THEN GO TO TOOBIG;
22   3         ELSE BUFF(PTR)=MINUS;  * INSERT MINUS SIGN IF IT FITS
23   3           PTR=PTR-1;
24   3           ENDIF
25   2         ENDIF
26   1       DO WHILE PTR GT 0;       * BLANK OUT ANY UNUSED PART OF BUFF
27   2         BUFF(PTR)=' ';
28   2         PTR=PTR-1;
29   2         ENDDO
30   1       RETURN
31   1  LABEL TOOBIG                  * CONVERTED NUMBER TOO BIG FOR BUFF
32   1       BUFF(1)=STAR;            * SET FIRST CHAR = * TO NOTE ERROR
33   1       RETURN
34   1     ENDPROC
35   0   END IFORM
 NO ERRORS DETECTED
```

```
1       * IREAD -- FUNCTION TO SCAN FOR AN INTEGER AND RETURN ITS VALUE
2       * CRB OCT 31, 1981
3            BEGIN IREAD;  ENTRY IREAD;
4            DCL I;                   * LICAL COPY OF IPTR
5            DCL L;                   * LENGTH OF LINE
6            DCL N;                   * VALUE OF NUMBER BEING CONVERTED
7            DCL PLUS=('+'),MINUS=('-'),SIGN;
8       *---------------
9       PROC IREAD(IPTR,LINE);
10   1       SIGN=0;                  * SET SIGN FOR POSITIVE NUMBER
11   1       N=0;
12   1       I=IPTR;                  * LOCAL COPY
13   1       L=LINE;                  * ASSUME LINE(0) CONTAINS LENGTH
14   1       DO WHILE LINE(I) EQ ' '; * TRIM LEADING BLANKS
15   2         IF I GE L; THEN IPTR=I; RETURN 0; ENDIF
16   2         I=I+1;
17   2         ENDDO
18   1       IF LINE(I) EQ PLUS; THEN GO TO IR1  ENDIF
19   1       IF LINE(I) EQ MINUS; THEN
20   2         SIGN=-1;               * SET SIGN FOR NEGATIVE NUMBER
21   2  LABEL IR1;
22   2         I=I+1;
23   2         DO WHILE LINE(I) EQ ' '; * ALLOW BLANKS AFTER + OR -
24   3           IF I GE L; THEN RETURN 0; ENDIF
25   3           I=I+1;
26   3           ENDDO
27   2         ENDIF
28   1       IF LINE(I) LT '0'; THEN RETURN 0; ENDIF
29   1       IF LINE(I) GT '9'; THEN RETURN 0; ENDIF
30   1       DO WHILE I LE L;
31   2         IF LINE(I) LT '0'; THEN EXIT ENDIF
32   2         IF LINE(I) GT '9'; THEN EXIT ENDIF
33   2         N=10*N-'0'+LINE(I);
34   2         I=I+1;
35   2         ENDDO
36   1       IF SIGN; THEN N=-N; ENDIF
37   1       IPTR=1;                  * MOVE INPUT POINTER TO TERMINATING CHAR
38   1       RETURN N;                * RETURN ACCUMULATED NUMBER AS VALUE
39   1     ENDPROC
40   0   END
 NO ERRORS DETECTED
```