NASA Contractor Report 159016

THE DESIGN OF RELATIVELY MACHINE-INDEPENDENT CODE GENERATORS

Robert E. Noonan

COLLEGE OF WILLIAM AND MARY
Williamsburg, Virginia 23185

NASA Contract NAS1-14972, Task 14
February 1979

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

FINAL REPORT

The Design of Relatively Machine-Independent Code Generators

Robert E. Noonan
Dept. Of Mathematics and Computer Science
College of William and Mary
Williamsburg, Va. 23185
(804) 253-4481

ABSTRACT

The goal of the research was to investigate the design of code generators which are relatively machine-independent. Two complementary approaches were investigated. In the first approach software design techniques were used to design the structure of a code generator for Halmat. The major result of this research was the development of an intermediate code form known as 7UP. The second approach viewed the problem as one in providing a tool to the code generator programmer. The major result of this investigation was the development of a non-procedural, problem oriented language known as CGGL (Code Generator Generator Language).

1. INTRODUCTION

This is a final report on the grant entitled "The Design of Relatively Machine-Independent Code Generators" conducted under contract to NASA Langley Research Center under contract NAS1-14972, task order 14.

A compiler for a programming language can be logically divided into the following phases: scanner, parser, code improver (or optimizer), and code generator. In the last 20 years a great deal of research has been expended on the first 3 phases, while little work has been done on the last phase. The design and implementation of code generators is widely thought to be an error-prone, expensive, highly machine-dependent task.

The goal of this research was to investigate the design of code generators which are relatively machine-independent. Specifically this means that for two machines with similar architectures (e.g., one-address with indexing and a single accumulator), large portions of the code generator would remain the same. Two separate but complementary approaches to this problem were investigated with fruitful results.

In the first approach software design techniques were used to design and iteratively improve the design of a code generator from the intermediate code HALMAT (for the language HAL/S <1975>) to the Intel 8080. The major result of this approach was the design of an intermediate code form known as 7UP. This design was implemented under separate contract by NASA to Computer Sciences Corp. (contract NAS1-14900). This work is described in the next section.

The second approach viewed the problem as one of providing a tool to the programmer who implements a code generator. The major result in this area was the development of a non-procedural, problem-oriented programming language known as CGGL (Code Generator Generator Language). An 8080 code generator was coded in CGGL and compared to the 7UP implementation. This research is discussed in the third section.

2. 7UP

In this part of the research various software design methodologies were to be used in the design of a code generator for the subset of HAL/S given in Figure 1. Specifically the techniques of Jackson <1975> and of hierarchical machine design <Dijkstra, 1972; Mills, 1971> were combined to produce an initial design. This initial

HAL/S	HALMAT Operators
integer constants	operand tag LIT
integer variables	operand tag SYT
integer expressions	operand tag VAC
subscripted variables	operand tag SREF
=, +, -	IASN, IADD, ISUB
indexing	DSUB
=, ^=, >, >=, <, <=	IEQU, INEQ, IGT, INGT, ILT, INLT
GO TO	BRA
labels	LBL
IF ...	IFHD (IF header)
THEN ...	FBRA (branch to ELSE)
ELSE ...	BRA (branch to end IF)
	LBL (ELSE label)
	LBL (end IF label)
DO WHILE	DTST (DO WHILE header)
	CTSTW (DO WHILE condition end)
	ETST (DO WHILE end)
DO UNTIL	DTST (DO UNTIL header)
	CTSTU (DO UNTIL condition end)
	ETST (DO UNTIL end)

Figure 1: HAL/S Subset

design is shown in Figure 2.

This design was implemented (under contract NAS1-14900 to Computer Sciences Corp.) using HALMAT as the intermediate code form and the Intel 8080 as the target machine. This implementation (in Pascal) was subject to a modularity

analysis <Myers, 1975>; the results of this analysis are summarized in Figure 3. Note that the term class in Figure 3 refers to a group of related modules. The code generator itself consists of approximately 40 routines.

In analyzing the module calling tree of this implementation, several problems became evident. The most serious of these was the lack of clear separation of levels, depicted in Figure 3 as classes 4 and 6 (accumulator management and IC generator). It was apparent that the internal code generated within the code generator should be made explicit.

Emphasis now shifted to the design of this internal code, which was named 7UP (or sometimes HAL/P). This internal code was to be closer to machine language than HALMAT and was to include accumulator management but not explicit addressing (that is, other than symbolic addressing). 7UP evolved into a hypothetical, one-address, single-accumulator machine. The operators and operands for this machine are shown in Figures 4 and 5.

The 7UP machine was analyzed and a number of recommendations were made. Some of these were aimed at eliminating redundant operations, while others were aimed at allowing as much local optimization as possible to be done in the HALMAT-to-7UP translation as possible. (Note that these recommendations were not implemented as of the date of this report). These revised 7UP operators are given in Figure 6.

The design goal for 7UP was to put as much of the work and local optimization as possible into the HALMAT-to-7UP translation and as little as possible into the 7UP-to-machine-language translation. The implementation was revised to explicitly generate and use 7UP. Although this implementation was not retargeted for a machine other than the Intel 8080, it was still felt that the goals were largely achieved. A modularity analysis of the revised implementation is given in Figure 7.

The goal of doing local optimization in the translation to 7UP was clearly perceived to make this translation ever more complex. It was desired to develop some form of tool to simplify this task. A programming language approach was adopted for this problem and the result of this research was the development of the language CGGL, which is described in the next section.

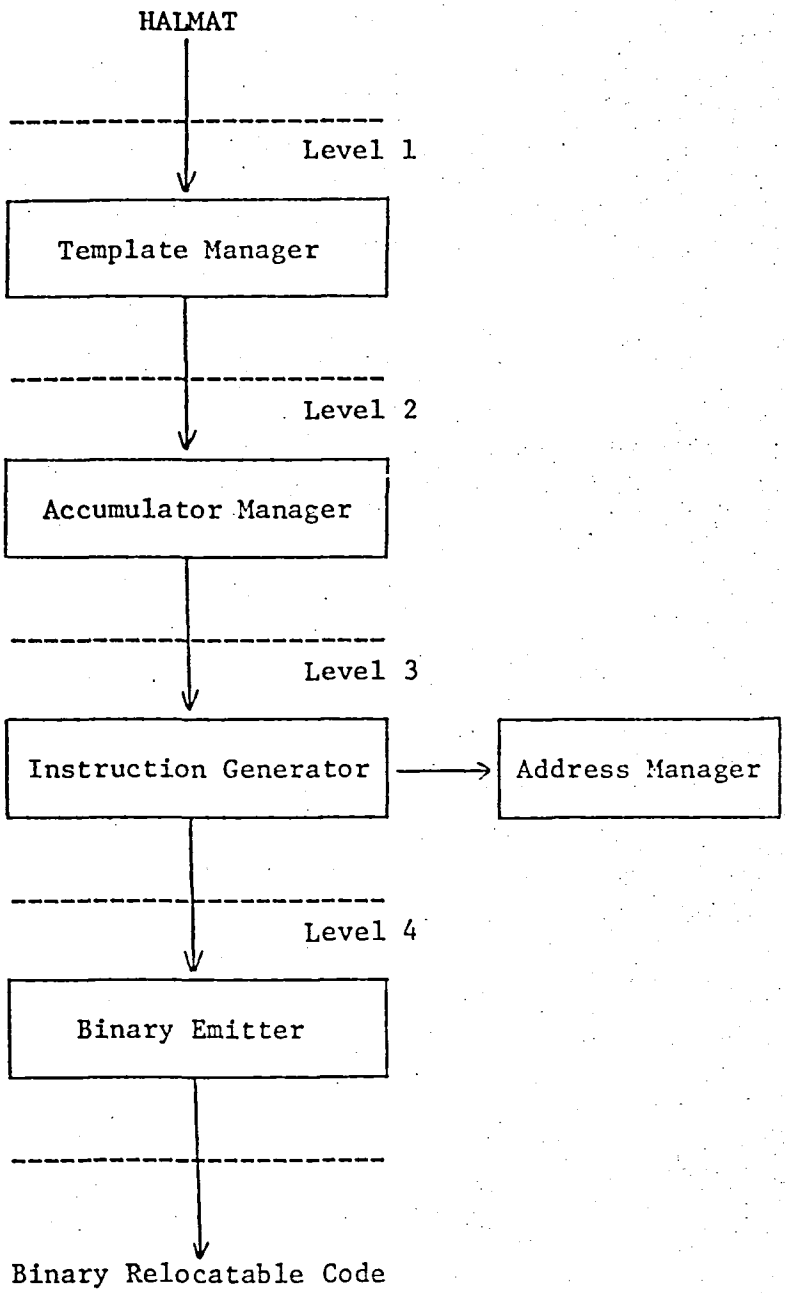

Figure 1: Preliminary Design of a Code Generator for INTEL 8080.

Class No.	Class No. / Class Name	1	2	3	4	5	6	7	8
1	8080 Support	3							
2	IC Support								
3	HALMAT Support			2					
4	Accumulator Management			2	1		2		
5	Storage Allocator	1	1						
6	IC Generator		3	2	2		4		
7	8080 Generator	3	2				2		
8	Template			6	3		3		
9	CODEGEN	1	2	3		3		2	1

Figure 3: Fan Out of Module Classes
Code Generator (v. 1)

Notation:

ACC = accumulator
EA = effective address (see Table 2)
PC = program counter
C(...) = contents of ...

Opcode	Mneumonic	Interpretation
0	STORE	C(ACC) → C(EA)
1	LOAD	C(EA) → C(ACC)
2	HALT	HAL/P machine halts
3	JUMP	EA → C(PC)
4	JFALSE	if C(ACC) = false then EA → C(PC)
5	JTRUE	if C(ACC) = true then EA → C(PC)
6	CALL	subroutine call
7	RETURN	return from a subroutine
8	ADD	C(ACC) + C(EA) → C(ACC)
9	SUB	C(ACC) - C(EA) → C(ACC)
10	MULT	C(ACC) * C(EA) → C(ACC)
11	CNEQ	(if C(ACC) ≠ C(EA) then true else false) → C(ACC)
12	CEQ	(if C(ACC) = C(EA) then true else false) → C(ACC)
13	CNGT	(if C(ACC) <= C(EA) then true else false) → C(ACC)
14	CGT	(if C(ACC) > C(EA) then true else false) → C(ACC)
15	CNLT	(if C(ACC) >= C(EA) then true else false) → C(ACC)
16	CLT	(if C(ACC) < C(EA) then true else false) → C(ACC)
17	INIT	program initialization
18	SUBSCR	C(ACC) + EA → C(ACC)
19	LOAD_IMD	EA → C(ACC)
20	ADD_IMD	C(ACC) + EA → C(ACC)
21	SAVE_ADDR	C(ACC) are saved in an address temporary
22	STORE_TEMP	C(ACC) → C(TEMP)
23	GEN_LABEL	associate the PC with the address field

Figure 4: Implemented 7up Operations

Tag Code	Mneumonic	Interpretation
1	SYT	Effective address determined by 8080 translators
2	INL	Internal label: address determined by LABEL operation
3	VAC	Temporary
4	XPT	
5	LIT	Literal: 8080 can use immediate instructions
6	IMD	Immediate: operand number is a constant
7	AST	
8	CSZ	
9	ASZ	
10	OFF	
11	SREF	Other than in a SAVE_ADDR operation indicates a subscripted reference
15	XREF	Address of an external procedure

Figure 5: 7up Operand Tags

Notation:

ACC = accumulator
EA = effective address (see Table 2)
PC = program counter
C(...) = contents of ...

Opcode	Mneumonic	Interpretation
0	STORE	C(ACC) → C(EA)
1	LOAD	C(EA) → C(ACC)
2	HALT	HAL/P machine halts
3	JUMP	EA → C(PC)
4	JFALSE	if C(ACC) = false then EA → C(PC)
5	JTRUE	if C(ACC) = true then EA → C(PC)
6	CALL	subroutine call
7	RETURN	return from a subroutine
8	ADD	C(ACC) + C(EA) → C(ACC)
9	SUB	C(ACC) - C(EA) → C(ACC)
10	MULT	C(ACC) * C(EA) → C(ACC)
11	CNEQ	(if C(ACC) ≠ C(EA) then true else false) → C(ACC)
12	CEQ	(if C(ACC) = C(EA) then true else false) → C(ACC)
13	CNGT	(if C(ACC) <= C(EA) then true else false) → C(ACC)
14	CGT	(if C(ACC) > C(EA) then true else false) → C(ACC)
15	CNLT	(if C(ACC) >= C(EA) then true else false) → C(ACC)
16	CLT	(if C(ACC) < C(EA) then true else false) → C(ACC)
17	INIT	program initialization
18	SUBSCR	C(ACC) + EA → C(ACC)
19	SAVE_ADDR	C(ACC) are saved in an address temporary
20	LABEL	associate the PC with the address field
21	PROC	associate the PC with the address field Generate code to save the return address.
22	INCR	C(EA) + C(ACC) → C(EA)
23	DECR	C(EA) - C(ACC) → C(EA)
24	CONST	indicates compile-time address calculation

Figure 6: Revised 7up Operations

Class No.	Class No. / Class Name	1	2	3	4	5	6	7	8	9	10
1	8080 Support	12							1		
2	7UP Support										
3	HALMAT Support			2					3		
4	Accumulator Management			2	1		2				
5	Storage Allocator	1							1		
6	7UP Generator						1		1		
7	8080 Generator	5	1					1	1		
8	Code Generator	3	1	1		3		2	1	1	
9	Miscellaneous			3							
10	Template			6	3		1		1		1
11	7UP Address Manager			2	2		1				

Figure 7: Fanout of Module Classes
(7up Version)

3. A CODE GENERATOR GENERATOR LANGUAGE

CGGL (pronounced sea-gull) is a non-procedural, problem-oriented language for writing code generators for compilers. The output from a CGGL compilation is a high-level language (in our case Pascal) program for generating machine code. CGGL is based on the work of Donegan <1973>.

Thus, CGGL is a language in which to express a program from which a code generator (CG) can be produced. The input to the latter is binary trees and the output machine or assembly code. In order to remain independent of the exact form of both the intermediate code (IC) and of the machine language, CGGL is used to generate only a portion (the Translate routine) of the actual code generator, as illustrated in Figure 8. The remaining routines must be

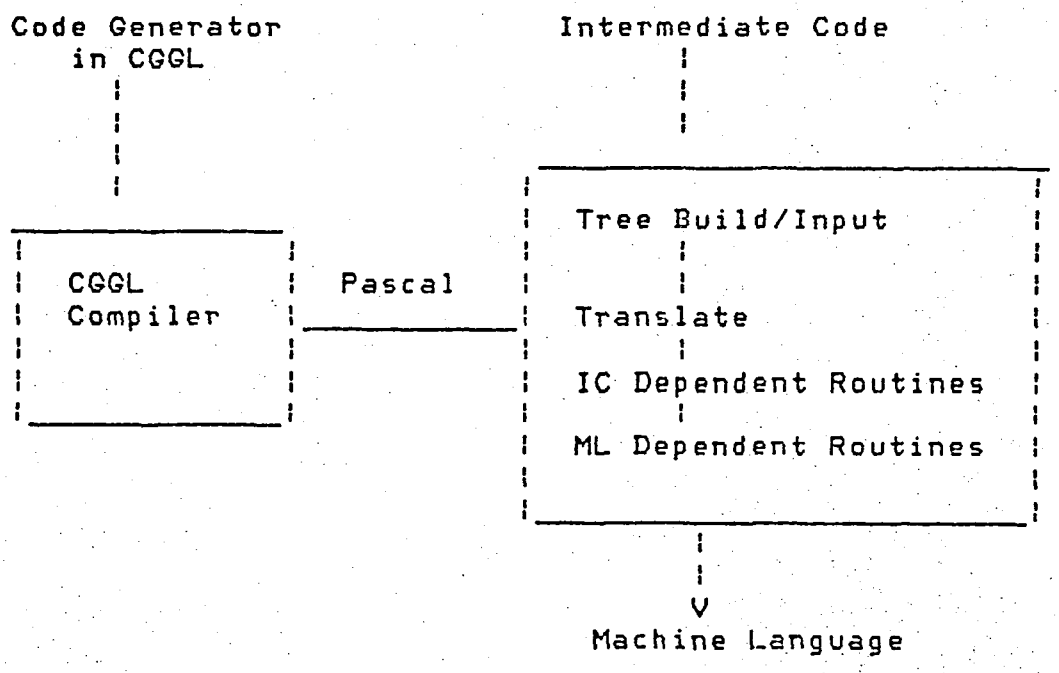

Figure 8: Role of CGGL in Code Generation

coded by hand in the CG language (in our case, Pascal).

A description of the language CGGL is given in Appendix A. The language itself can be used in a variety of ways. For example, it could be used to translate from HALMAT to 7UP and from 7UP to machine code. As an experiment it was decided to translate HALMAT directly to 8080 machine code using CGGL. This code generator is given in Appendix C.

As a point of comparison, it was decided to compare the quality of code generated in the 7UP implementation versus the CGGL implementation. (Note at that time no CGGL compiler existed). CGGL generated code which used considerably less time and space than 7UP. These results are given in Figure 9. It should be noted that at the time the 7UP implementation was done, the quality of code was not a consideration, only the speed with which the

	7UP		CGGL	
HAL/S	Bytes	Cycles	Bytes	Cycles
x = 1	8	34	5	20
x = x + 1	12	51	8	33
x = x + y	12	51	10	43
... x - (y + z) ...	28	1191	25	1251
IF x = y + 1 THEN	35	1381	24	7
x$(3) = 02	21	041	14	8
x = y$(i - 1)	58	2501	67	2

Figure 9: Comparison of 7UP and CGGL

implementation could be completed.

In addition the CGGL implementation was considerably faster to code and easier to modify. One indication of this was the ease with which subscripting (indexing) was added. In the 7UP implementation this caused considerable problems, since the 8080 lacks true index registers. The basic addressing mechanism had to be substantially changed as well as the accumulator management routines. However, in the CGGL implementation only one new operation (DSUB itself), two new states were added to condition AROP, three new transitions were added to transition AROP, and three new terminal configurations to operation IASN. Unlike the 7UP implementation, substantial changes to previous work was not required. Thus, for assignments or expressions not involving indexing, the same code is generated as previously (that is, before the addition of subscripting). In addition, unlike 7UP 16-bit temporaries were not introduced.

Another example of the ease with which a CGGL program can be modified was the introduction of immediate instructions. This change involved modifying one transition and adding two terminal configurations.

In conclusion, implementing a HALMAT (or any other IC) code generator in CGGL is quite straightforward for a single accumulator machine. CGGL is best at expressing the complex case analysis required to generate good local code. Experience (admittedly limited) has shown that code

generators expressed in CGGL are easily modifiable to add new operators, to improve the code generated, and to fix bugs.

4. CONCLUSIONS

The research into the application of software design methodologies to the design of code generators took some surprising directions. After an initial design had been implemented, a modularity analysis showed problems in the design. This lead to the development of an internal form called 7UP which is closer to machine language than HALMAT. This internal form is expected to decrease the effort necessary to retarget a HAL/S compiler.

As the investigation into implementing a larger subset of HAL proceeded, it became clear that the code generator would become not only larger, but also more complex. The investigation lead to the development of the non-procedural language CGGL.

A code generator for the Intel 8080 was written in this new language. However, the code generator could not be tested because of the lack of a CGGL compiler. Preliminary experience indicates that CGGL may have a revolutionary impact on the development of code generators.

5. REFERENCES

1. Dijkstra, E. W. Notes on structured programming. Structured Programming, by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Academic Press, 1972.

2. Donegan, M. K. An Approach to the Automatic Generation of Code Generators. Ph.D. Thesis, Rice University, 1973.

3. HAL/S Language Specification. Technical Report IR-61-6, Intermetrics, 1975.

4. Jackson, M. A. Principles of Program Design. Academic Press, 1975.

5. Mills, H. Top down programming in large systems. Debugging Techniques in Large Systems, ed. by R. Rustin. Prentice-Hall, 1971.

6. Myers, G. J. Reliable Software through Composite Design. Petrocelli/Charter, 1975.

APPENDIX A: CGGL

A CGGL program consists of conditions, variables, transitions, conflicts, operations, and procedures. The elements of a CGGL program can be arranged in any convenient order, although the compiler may insist on a specific order. A complete grammar for CGGL is given in Appendix B.

The input to the code generator generated is assumed to be an IC tree consisting of operators and operands (end nodes). The exact description of a node is independent of CGGL.

A condition statement specifies a named class of states which an operand can take on. For example:

```
condition AROP = input LIT, VAC, SYT
    internal INACC, INREG, ONSTACK;
```

says that an operand of type LIT, VAC, or SYT belongs to the class AROP. In addition, three internal states are defined. All of these states are mutually exclusive, i.e., the operand can be in at most one of them. The condition statement

```
condition FLOW = input INL;
```

says that an operand of type INL belongs to the class FLOW; this class has no other input or internal states. States are used in CGGL to keep track of the status of a given operand, for example, whether an operand is in the accumulator INACC.

Variables in CGGL are used to pass information down the tree. Such variables may be used to control decisions in both transitions and operations. For example:

```
var CMP_MODE = (ASSIGN_CMP, JMP_T, JMP_F);
```

declares a variable with three possible values; one of these ASSIGN_CMP is its initial value. One method of changing the value of a variable is by means of the let statement, which can be used only within an operation. For example, consider:

```
condition FBRA (FLOW, AROP);
    let CMP_mode = JMP_F; ...
```

On entry to this operation, the current value of CMP_MODE is saved and is then replaced by the value JMP_F; just before exit, the old value is restored. The values of variables can also be changed by transitions, as will be seen.

A transition statement defines all of the state changes which can occur on a given, named condition. A state change X -> Y specifies how an operand moves from an input or internal state X to an internal state Y. For example, consider:

```
transition AROP;
    LIT -> INACC: GEN2(MVI_A, #);
```

This transition specifies what code is required to move a literal into the accumulator for an INTEL 8080. The symbol # stands for the operand in question. GEN2 is a call on a Pascal routine for generating 2 byte instructions for the INTEL 8080. Thus, transitions associate (Pascal) code to be generated with state changes on operands.

Since certain operations, e.g. indexing on the 8080, may also require the use of the accumulator, CGGL allows the explicit statement of conflicting states for pairs of operands, for example:

```
conflict AROP, AROP;
    INACC, INACC;
```

The above example states that if you have one operand in the accumulator, it is incorrect to put the other operand into the accumulator. All such conflicts are assumed to be commutative. Note that the pair ONSTACK, ONSTACK is not a conflict.

Another, more subtle way of specifying conflicts is through the use of variables and transitions. Consider the following:

```
var ACC_STATUS = (ACC_FREE, ACC_BUSY);
transition AROP using ACC_STATUS;
    LIT, ACC_FREE -> INACC, ACC_BUSY: GEN2(MVI_A, #);
```

This specifies that if one wants to load a literal into the accumulator, the value of ACC_STATUS must be checked and if necessary, the accumulator freed (by means of another transition). Although unnecessary for the single accumulator machine, such an approach appears necessary for multiple accumulators in order to prevent an explosion of states.

An operation specifies for each operator the number of arguments and condition class of each argument and the internal state (if any) computed by the operation. The operation lists terminal configurations and their associated code generator statements. Possible transitions are entirely determined by the transitions given for each condition class and by the terminal states allowed. For example:

```
operation IADD(AROP, AROP) returns AROP;
    INACC, ADDR_LOADED -> INACC: GEN1(ADD_M);
    ADDR_LOADED, INACC -> INACC: GEN1(ADD_M);
```

specifies that IADD has two operands of condition class AROP, leaves its result in the ACC (state INACC), and requires that one of its arguments be INACC and the other be ADDR_LOADED. The operation IADD is easily seen to be commutative. Like transitions, operations may also use global variables, as for example:

```
operation IEQU(AROP, AROP) using CMP_MODE returns
AROP;
    INACC, ADDR_LOADED, JMP_T -> INACC:
    GEN1(CMP_M), GEN3(JZ, JMP_TARGET);
```

It should be noted that the returns clause as well as its associated state on the right-hand-side of an -> is entirely redundant. This information has already been specified in a transition.

A procedure specifies a group of statements. It is useful for specifying long sequences of code. A procedure call is specified by a call <proc name>. An example of a procedure is the following:

```
proc CMP_EQ;
   GEN2(MVI_A, ONE), GEN3(JZ, PC+4), GEN1(ZAC);
```

APPENDIX B: CGGL SYNTAX

```
<CGGL program>   ::= <statement list> eof
<statement list> ::= <statement list> <statement> |
                     <statement>
<statement>      ::= <var decl> | <condition> |
                     <transition> | <conflict> |
                     <operation> | <proc>


<var decl>       ::= var <id> = ( <list> ) ;



<condition>      ::= condition <id> = input <list>
                        <internal part> ;
<internal part>  ::= internal <list> | e


<transition>     ::= transition <list> <using part> ;
                        <test set> end tr ;


<conflict>       ::= conflict <conflict list> end co ;
<conflict list>  ::= <conflict list> <list> ; | <list>


<operation>      ::= operation <id> ( <list> )
                        <using part> <return part> ;
                        <assign list> <test set> end op ;


<assign list>    ::= <assign list> <assign> | e
<assign>         ::= let <id> = <id> ;


<proc>           ::= proc <id> ; <PL code> ; end pr ;


<using part>     ::= using <list> | e
<returns part>   ::= returns <id> | e


<test set>       ::= <test set> <test> | <test>
<test>           ::= <list> <result> : <PL code> ;
<result>         ::= -> <list> | e


<PL code>        ::= ' <text> '


<list>           ::= <list> , <id> | <id>
```

Remarks

1. An <id> can be composed of letters, digits, and underscores, the first of which must be a letter.
2. A comment may occur anywhere a blank may occur and is defined as:

   <comment> ::= (* <text> *)
3. Certain characters in the reference language require special treatment in the implementation. All underlined words such as var are reserved.
4. # in <text> is replaced by the operand name OP.

```
OPERATION ISUB(AROP, AROP) RETURNS AROP;
  IN_ACC, LIT -> IN_ACC: 'GEN2(SUB_I, #2)';
  IN_ACC, ADDR_LOADED -> IN_ACC: 'GEN1(SUB_M)';
  IN_ACC, IN_REG -> IN_ACC: 'GEN1(SUB_C)';
END_OP;

OPERATION DSUB(AROP, AROP) RETURNS AROP;
  SYT, IN_ACC -> IN_ACC: 'GEN1(MOV_C_A); GEN2(MVI_B, ZERO);
                          GEN3(LXI_HL, #1); GEN1(DAD_BC)';
END_OP;

OPERATION IEQU(AROP, AROP) USING CMP_MODE RETURNS AROP;
  IN_ACC, ADDR_LOADED, JMP_T -> IN_ACC:
        'GEN1(CMP_M);
         GEN3(JZ, JMP_TARGET)';
  ADDR_LOADED, IN_ACC, JMP_T -> IN_ACC:
        'GEN1(CMP_M);
         GEN3(JZ, JMP_TARGET)';
  IN_ACC, IN_REG, JMP_T -> IN_ACC:
        'GEN1(CMP_C);
         GEN3(JZ, JMP_TARGET)';
  IN_ACC, ADDR_LOADED, JMP_F -> IN_ACC:
        'GEN1(CMP_M);
         GEN3(JNZ, JMP_TARGET)';
  ADDR_LOADED, IN_ACC, JMP_F -> IN_ACC:
        'GEN1(CMP_M);
         GEN3(JNZ, JMP_TARGET)';
  IN_ACC, IN_REG, JMP_F -> IN_ACC:
        'GEN1(CMP_C);
         GEN3(JNZ, JMP_TARGET)';
END_OP;

OPERATION IGT(AROP, AROP) USING CMP_MODE RETURNS IN_ACC;
  IN_ACC, ADDR_LOADED, JMP_T -> IN_ACC:
       'GEN1(CMP_M);
        GEN3(JZ, PC+2);
        GEN3(JNC, JMP_TARGET)';
  IN_REG, IN_ACC, JMP_T -> IN_ACC:
       'GEN1(CMP_C);
        GEN3(JC, JMP_TARGET)';
  ADDR_LOADED, IN_ACC, JMP_T -> IN_ACC:
       'GEN1(CMP_M);
        GEN3(JC, JMP_TARGET)';
  IN_ACC, ADDR_LOADED, JMP_F -> IN_ACC:
       'GEN1(CMP_M);
        GEN3(JZ, JMP_TARGET);
        GEN3(JC, JMP_TARGET)';
  ADDR_LOADED, IN_ACC, JMP_F -> IN_ACC:
       'GEN1(CMP_M);
        GEN3(JNC, JMP_TARGET)';
  IN_REG, IN_ACC, JMP_F -> IN_ACC:
       'GEN1(CMP_C);
       GEN3(JNC, JMP_TARGET)';
END_OP;
```

```
OPERATION BRA(FLOW) RETURNS FLOW;
  INL -> INL:   'GEN3(JMP, #1)';
END_OP;

OPERATION LBL(FLOW) RETURNS FLOW;
  INL -> INL:   'SET_ADDR(#1, PC)';
END_OP;

OPERATION IFHD(FLOW) RETURNS FLOW;
  INL -> INL:   '';
END_OP;

OPERATION DTST(FLOW) RETURNS FLOW;
  INL -> INL:   'PUSH_DOSTACK(PC, #1)';
END_OP;

OPERATION ETST(FLOW) RETURNS FLOW;
  INL -> INL:   'POP_DOSTACK(OLD_PC);
                 GEN3(JMP, OLD_PC)';
END_OP;

OPERATION FBRA(FLOW, AROP) RETURNS FLOW;
  LET CMP_MODE = JMP_F;
  INL, VAC-> INL: 'BLD_HALMAT(JMP_TARGET, #1);
             TRANSLATE(#2)';
END_OP;

OPERATION CTSTW(AROP) RETURNS FLOW;
  LET CMP_MODE = JMP_F;
  VAC -> INL: 'TOP_DOSTACK(JMP_TARGET);
       TRANSLATE(#1)';
END_OP;

OPERATION  CTSTU(AROP) RETURNS FLOW;
  LET CMP_MODE = JMP_T;
  VAC -> INL:   'TOP_DOSTACK(JMP_TARGET);
        TRANSLATE(#1)';
END_OP;
;
```

ACKNOWLEDGEMENTS

I would like to acknowledge the help and support of Dr. Terry Straeter and Dr. John Knight of the NASA Langley Research Center in this research. I would also like to thank Patti Timpanaro of Computer Sciences Corp. for her work in helping to design the 7UP code generator and for her work in implementing it.