PROCEEDINGS
SIXTH HAWAII INTERNATIONAL CONFERENCE
ON SYSTEM SCIENCES

Edited By

Art Lew

Conference Held

January 9, 10, 11, 1973
University of Hawaii
Honolulu, Hawaii

Sponsored By:
Department of Electrical Engineering and
Department of Information and Computer Sciences
University of Hawaii

Supported By:
U.S. Army Research Office, Durham

In Cooperation With:
The IEEE Computer Society
The IEEE Control Systems Society
The IEEE Groups on
Circuit Theory
Systems, Man and Cybernetics
The Hawaii Section of IEEE
Simulation Councils, Inc.
Society for Industrial and Applied Mathematics

META S: A METALANGUAGE FOR COMPILERS†

W. J. Chandler

Computer Science Program
University of Southern California
Los Angeles, California 90007

Abstract

A metalanguage for efficient description of one-pass compilers is presented. Its input is the conventional form of BNF productions with interlaced actions and tests. META S differs from other metalanguages by its pre-defined data structures, arithmetic capability, and symbol table with arbitrary attributes.

1. INTRODUCTION

During the preceding decade, a succession of compiler metalanguages have been developed [1,2,3,4,5,6]. All of these share the feature that their input is essentially a set of Bakus-Naur Form (BNF) productions with interspersed actions and tests. The power of these METAlanguages varies considerably. The earliest one, META 2 [5], only allowed output commands as an action; while the later ones (META 3 [6], META 5 [3], META PI [2], TREE META [1], and META 7 [7]) have a menu of actions which can be performed on internal data structures. Some allow full back-up while others have limited or no back-up. Some allow user-defined data structures, while others have only pre-defined structures. A general familiarity with metalanguages is assumed.

2. GENERAL ORGANIZATION

META S is introduced as a metalanguage for efficient and concise descriptions of one-pass translations (while multi-pass translations can be written in META S, this would result in inefficient and artificial descriptions). As in other META languages, the input to META S is a set of BNF productions with interlaced actions and tests. Back-up is not provided because it is unnecessary for one-pass translations, and overhead for back-up is costly when complex data structures are involved.

Pre-defined data structures in META S consist of pushdown stacks, global variables, and a symbol table. There is a main stack (designated by *) which is used for communication between the recognizer and actions in the BNF productions. There are 9 other stacks (designated by * followed by a digit) which can be used for any purpose. That is, *1 refers to stack 1, *2 refers to stack 2, etc. *0 means the same as *, and refers to the main stack. Modifications to a stack can occur only at its top. However, reference to interior stack elements is provided by following a stack designation with an integer enclosed in parentheses. That is, *1(3) refers to the third element of stack 1 (the topmost element in the zero element). Also, there are 26 global variables (designated by single letters A through Z) which can be used for any purpose.

A symbol table is provided which contains identifiers, strings, and numbers. Each symbol table entry has three attributes, designated by digits 1, 2, and 3. There is no a priori interpretation of these attributes; any item which can be manipulated by META S can be assigned to one of these attributes. For example, the top element of a stack can be assigned as the third attribute of a symbol table entry. The recognizer automatically inserts identifiers, numbers, strings, etc. in the symbol table.

3. SYNTAX

The syntax of META S is given by the following self-description which is a translation from META S to the META S machine. (See Figures 1 and 2.)

```
.SYNTAX PROGRAM;
PROGRAM = '.SYNTAX' .ID ';'
    .OUT('CALL',*/,'DIAG'/,'HALT')
    STMT $ STMT '.END' ';' ;
```

A program consists of the keyword .SYNTAX, followed by an identifier which is the start symbol of the BNF productions, followed by one or more statements, and terminated by the keyword .END.

†This work was supported by the Joint Services Electronics Program (U. S. Army, U. S. Navy, and U. S. Air Force) under Grant No. F44620-71-C-0067.

```
STMT = . ID . OUT(*) '=' EXP1 ';' ;
```

A statement is an identifier (which is the name of a syntactic type), followed by =, followed by an expression-1 (which defines the syntactic type), and terminated by a semi-colon.

```
EXP1 = EXP2 . LGEN(*1)
     $('/' . OUT(, 'BT', *1(0))EXP2) . OUT(*1) ;
```

An expression-1 is one or more alternatives (each of which is an expression-2) separated by slashes.

```
EXP2 = EXP3
```

An expression-2 is an expression-3,

```
     /ACTION$ACTION
     (EXP3/. TRUE. OUT(, 'SET'));
```

or a sequence of one or more actions, possible followed by an expression-3.

```
EXP3 = TEST . LGEN(*1) . OUT(, 'BF', *1(0))
     $(ACTION/TEST. OUT(, 'DIAG')). OUT(*1);
```

An expression-3 is a test followed by a sequence of actions and tests.

```
TEST = '(' EXP1 ')' /. STRING . OUT(, 'TEST', *)
```

A test is an expression-1 enclosed in parentheses, or a literal string.

```
     /. TNAMEO . OUT(, 'EXEC', *)
     /. TNAME. OUT(, 'EXEC', *)'('ARGLIST')';
```

or a test function possibly with an argument list.

```
ACTION = '$' . LGEN(*1) . OUT(*1(0))
     TEST . OUT(, 'BT', *1/, 'SET')
```

An action is a test which is repeated until it fails,

```
     /. ANAMEO . OUT(, 'EXEC', *)
```

or an action function without arguments,

```
     /. ANAME . OUT(, 'EXEC', *) '(' ARGLIST ')'
```

or an action function followed by an argument list,

```
     /'. OUT' . OUT(, 'EXEC', 'OUT')
       '(' OUT1 $ OUT1 ')' . OUT(, 'END LIST')
```

or a call to the output function followed by a sequence of elements to be printed,

```
     /'. CGEN' . OUT(, 'EXEC', 'CGEN')
       '(' SETITEM ', ' ARITHEXP ')'
       . OUT(, 'END LIST');
```

or a call to the arithmetic expression function followed by suitable arguments.

```
OUT1 = '/'
```

An out-1 is a slash denoting end of record,

```
     /','
```

or a comma meaning to skip to next tab position

```
     / ITEM;
```

or an item to be printed.

```
ARGLIST = ITEM $(',' ITEM) . OUT(, 'END LIST');
```

An argument list is a sequence of items separated by commas.

```
ITEM = . STRING . OUT(, *)/ELEMENT;
```

An item is a string or an element.

```
ELEMENT = '*'(. DIGIT/. TRUE. PUSH('0', *))
     ('(' . NUMBER ')' . POP(*, A)
     . OUT(, 'ST', *, '('A')')
     /. TRUE . OUT(, 'ST', *))
```

An element is a reference to one of the ten stacks,

```
     /. LETTER . OUT(, 'GV', *)
```

or a reference to one of the 26 global variables.

```
     /. NUMBER . OUT(, *);
```

or an unsigned integer.

```
SETITEM = '*'(. DIGIT/. TRUE. PUSH('0', *)
     . OUT(, 'ST', *)
```

A set item is a reference to one of the ten stacks,

```
     /. LETTER. OUT(, 'GV', *);
```

or a reference to one of the 26 global variables.

```
ARITHEXP = TERM $('+' TERM. OUT(, '+')
     /'-' TERM . OUT(, '-'));
```

An arithmetic expression is a sequence of terms separated by additive operators.

```
TERM = FACTOR $('*' FACTOR . OUT(,'*')
     /'/' FACTOR. OUT(, '/'));
```

A term is a sequence of factors separated by multiplicative operators.

```
FACTOR = PRIMARY/'-' PRIMARY. OUT(, 'U-');
```

A factor is a primary, possibly preceded by a unary minus sign.

```
PRIMARY = ELEMENT/'(' ARITHEXP ')';
```

A primary is an element or an arithmetic expression enclosed in parentheses.

```
. END;
```

Terminates the program.

4. FUNCTIONS

META S contain several built in tests and actions which operate on interval data structures. Actions are not permitted to modify stack interiors. For example, the argument of . LGEN cannot be *1(2). Each of the test functions sets the mode flag according to its result.

TESTS

. ID Tests input for an identifier. Inserts it into symbol table, and onto main stack.

. NUMBER Tests input for an unsigned integer. Inserts it into symbol table, and onto main stack.

. STRING Tests input for a sequence of characters enclosed in quote marks. Inserts it into symbol table, and onto main stack.

. LETTER Tests input for a letter, and pushes it onto main stack.

. DIGIT Tests input for a digit, and pushes it onto main stack.

. TRUE This test is always satisfied.

. CONS($a_1, \cdots, a_n$) Determines if all of the a_i contain numbers.

. TEMP(a_1) Determines if a_1 contains a name generated by . TGEN.

. EQ($a_1, \cdots, a_n$) Determines if the contents of a_i are all identical.

. NE($a_1, \cdots, a_n$) Complement of . EQ($a_1, \cdots, a_n$).

. TNAMEO Tests input for name of one of the test functions without arguments (. ID, . NUMBER, etc.). Pushes name onto main stack.

. TNAME Tests input for name of one of the test functions with arguments (. CONS, . TEMP, . EQ, . NE). Pushes it onto main stack.

. ANAMEO Tests input for name of one of the action functions without arguments (. STPRNT, . TCLR). Pushes it onto main stack.

. ANAME Tests input for name of one of the action functions with arguments (. LGEN, . TGEN, . PUSH, . POP, . AGET, . ASET, . INSERT). Pushes it onto main stack.

ACTIONS

. STPRNT Prints the symbol table.

. LGEN (a_1) Creates a unique "label" name, and puts it into a_1.

. TGEN (a_1) Creates a unique "temporary" name, and inserts it onto a_1.

. TCLR Resets the generator used by . TGEN.

. PUSH($a_1, \cdots, a_n$) Takes one element from each of a_1 through a_{n-1}, and inserts them onto a_n ($n \geq 2$).

. POP ($a_1, \cdots, a_n$) Takes n-1 elements from a_1 and inserts them into $a_2, a_3, \cdots, a_n$ ($n \geq 2$).

. AGET(a_1, a_2, a_3) Obtains the a_2 attribute of a_1 and inserts it into a_3.

. ASET(a_1, a_2, a_3) Sets the a_2 attribute of a_1 to the contents of a_3.

. CGEN(a_1, a_2) Evaluates arithmetic expression a_2, inserts result into symbol table and a_1.

. INSERT($a_1, \cdots, a_n$) Inserts into symbol table and a_1, the identifier obtained by concatenating contents of a_2 through a_n ($n \geq 2$).

5. REFERENCES

1. Carr, S. B., Luther, D. A., and Erdman, S., "The TREE-META Compiler-Compiler System," RADC-TR-69-83, University of Utah, March 1969.

2. O'Neill, J. T., "Meta Pi – An Interactive On-Line Compiler-Compiler," Proc. AFIPS 1968 FJCC, vol. 33, pp. 201-218.

3. Oppenhiem, D. K., "The Meta 5 Language and System," TM 2396, System Development Corproation, Santa Monica, Calif. January 1966.

4. Schneider, F. W. and Johnson, G. P., "META 3-A Syntax Directed Compiler Writing Compiler to Write Efficient Code," Proc. 19th ACM Nat'l. Conf., 1964.

5. Schorre, D. V., "Meta 2: A Syntax Oriented Compiler Writing Language," Proc. 19th ACM Nat'l. Conf., 1964.

6. Tyrrill, A. R., "The Meta 7 Translator Writing System," Report UCLA-ENG-7122, University of California at Los Angeles, School of Engineering.

FIGURE 1

INSTRUCTIONS OF META S MACHINE

CALL R	Recursive branch to item labeled R.
B R	Unconditional branch to item labeled R.
BF R	Branch to R if mode flag is false.
BT R	Branch to R if mode flag is true.
TEST S	Determine if string S occurs next in the input; set mode flag accordingly.
RETURN	Branch to instruction following the most recent CALL.
SET	Set mode flag to true.
HALT	Halt
DIAG	If mode flag is true, then perform diagnosis function and halt, else continue.
EXEC R	Perform function R; an argument list may follow this instruction.

FIGURE 2

ARGUMENT LIST COMPONENTS

ST i (j)	Reference to j-th element of stack i; stack i is not altered; j = 0 refers to top element
ST i	Reference to top element of stack i; stack i is either popped, or an item is pushed onto it.
GV a	Reference to global variable a.
Literal	This may be a string, an unsigned integer, or a letter.
operator	One of the arithmetic operators +, -, /, *, or unary minus. Arithmetic expressions appear in reverse polish notation.
END LIST	Last entry in argument list.