CGTM # 102
David Gries
November 1967

COMPILER IMPLEMENTATION SYSTEM

The syntax sublanguage

1. Introduction

Each phase of the compiler may have a syntax subprogram; it must have a semantic subprogram. The purpose of the syntax subprogram is to "parse" part or all of the source program being compiled. The first syntax sublanguage to be implemented will be "production language" [1]. Other parsing techniques will be implemented later.

2. The main stack

Basic to a phase with a syntax subprogram is a MAIN STACK (see CGTM 101), which is STATIC (for efficiency purposes) and which must be declared in the semantics subprogram. This stack is needed (usually) in order to parse the input correctly; it also serves as the main communication between syntax and semantics. A main stack may be global to several phases or passes.

The structure of the stack elements may be determined by the compiler writer, with the following restrictions. The structure definition which defines the elements must begin with

STRUCTURE <identifier> (BYTE2, BYTE2, BYTE2

Of course, the compiler writer may name the components if he wishes. The first component has internal uses.

The syntax and semantic programs of a phase perform complementary functions with regard to the main stack. The syntax program implements the analysis of syntax by manipulating the syntax portion (2^{nd} component) of each element; the semantic program updates the semantic portion (3^{rd}

component and any others the compiler writer wishes to use) while performing functions such as generating or optimizing the object program or checking for semantic errors.

3. Processing of input text

A phase makes a single scan through the symbols in (part of) the input text, or through the symbols produced by another phase or pass. These symbols fall into four classes:

(1) those which the syntax program will recognize as reserved words (such as +, /, (, BEGIN, etc. in ALGOL). These words are part of the language in which the source program is written.

(2) numbers.

(3) all other symbols, classed as identifiers.

(4) identifiers defined in the compiler as INT (only when processing output of a preceding phase or pass).

The definition of reserved words and specifications for the formation of numbers and identifiers are specified CGTM 103 and are not discussed here.

When a reserved word is "scanned", a 16 bit representation of this symbol is put in a specific location named SCANSYM, and pushed onto the main stack (2^{nd} component). When an identifier or number is scanned, an item denoting "identifier" or "number" is entered instead, along with a 16 bit representation of the identifier or number in the semantic portion (3^{rd} component).

Following a scan operation, the top few syntax entries and the last symbol scanned determine in some manner a set of routines to execute. These routines may modify the syntax portion of the stack, jump to semantic routines, record error messages, cause another symbol to be scanned, etc.

The process of scanning and matching to determine which routines to execute continues until all the symbols have been scanned or until the phase is "turned off" or terminated for some reason. The syntax program specifies the technique to be used in determining from the stack and scanned symbol which routines to execute. The following sections present the syntax and semantics of this syntax meta-language.

4. Syntax of the syntax sublanguage

<syntax> ::= PRODLANG <syntax dec>* PRODUCTIONS <production>* ENDSYNTAX

<syntax dec> ::= INT <identifier>*

::= CLASS <class name> <symb>*

::= CLASSLAB <class name> <symb> <semantic label>*

<production> ::= <label>: <symbl>* > <symbl>* <action>

<action> ::= EXEC <semantic label> | GO <label> | SCAN

::= SCAN <unsigned integer> | HALT <unsigned integer>

::= CALL <label> | RETURN

::= ERROR <unsigned integer> | WHEN SIGNAL GO <label>

::= UNSTK <unsigned integer>

::= STAK <unsigned integer> | STAK <identifier> | STAK SCANSYM

<class name> ::= <identifier>

<label> ::= <identifier>

<semantic label> ::= <identifier>

<symb> ::= (See Section 4.b below)

<symbl> ::= (See Section 4.c below)

5. Semantics of the syntax sublanguage

a) INT The <identifiers> declared as INT can be thought of as "nonterminal symbols" of the source language. They are entries to be made in the syntax portion (second component) of the main stack. Each <identifier> is represented by a 16 bit number (assigned by the metacompiler).

b) CLASS and CLASSNUM

A <symb> is

1) Any identifier previously declared INT.

2) The symbols I (representing "identifier")
and N (representing "number")

3) Any reserved word of the source language(which does not begin with "$") except INT, CLASS, CLASSLAB, and PR$DUCTI$NS.

4) $INT, $CLASS, $CLASSLAB, PRDUCTI$NS, and $ any reserved word of the source language beginning with $. The "$" is an escape symbol indicating that the following sequence of characters up to the next blank is a reserved word of the source language.

CLASS <identifier> (class name) is simply a notational convenience; a production containing a class name is equivalent to a

sequence of productions containing each of the corresponding <symb>s.

In the CLASSLAB declaration, the <semantic label>s must be labels (which are not in procedures) in the associated semantic sublanguage program. Used in productions, its function is the same as CLASS declaration, with an extra convenience mentioned later in discussing <action>s.

c) <production>s

A <symbl> is

1) Any identifier previously declared as INT, CLASS or CLASSLAB.

2) The symbols I (representing "identifier")

N (representing "number")

ANY (this matches any symbol; see below)

3) Any reserved word of the source language, except one containing the characters ">" or ":"

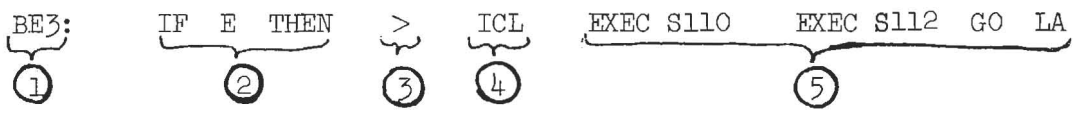
4) Any reserved word of the source language, except the reserved words appearing in <action>s (see syntax), and ENDSYNTAX.

Suppose "RETURN" and ">E" are source language symbols. They may not appear in the productions as such. One should declare two classes, say

```
CLASS  RET  RETURN
CLASS  LELE  >E
```

and use the names RET and LELE in the productions. Since there is only one element in each class, there will be no loss of efficiency.

The syntax and semantics of productions are described most easily by an example. The following production is typical.



(1) BE3 is a label for this production. (1) is optional,

(2) "IF E THEN" is an expected stack configuration. IF and THEN are reserved words, E (denoting "expression") is an internal symbol (declared as INT). (2) always contains 1 to 5 reserved words, class names, or internal symbols. (2) is matched against the top few syntactic elements (2nd component) of the stack. If no match occurs, the next production is checked, and so on, until a match occurs. (IF (2) contains a class name, then a match occurs for that position if the corresponding stack position contains a reserved word or internal symbol in that class).

Let us suppose that a match occurs: the top three stack (syntactic) elements are IF, E and THEN. The following occurs.

(3) ">" indicates that the top stack elements matching (2) are to be replaced by (4). In this case the top three entries are removed and ICL is pushed onto the stack in the syntax word. Note that the semantic portion (3rd, 4th, ... , components) of the stack is not disturbed. If (3) is absent (4) must also be absent and no alteration is made.

(4) contains 0 to 3 reserved words, internal symbols, or class names. If (4) contains a class name, (2) must contain the same class name. The actual symbol in the stack corresponding to the topmost (rightmost) position of (2) with that class name is inserted, and not the class name itself.

(5) EXEC S110, EXEC S112 and GO LA are known as <u>actions</u>. S110 and

S112 are labels in the semantic program of this phase (written in the semantic sublanguage). The semantic sublanguage will provide primitives for manipulating the semantic portions of the stack. EXEC S110 causes transfer of control to the semantic program at label S110. Upon return, GO LA causes the next comparison of stack with productions to begin at the production labeled LA. The absence of a "GO" action means that the stack will be compared with the succeeding production when all actions in this line have been executed.

Other actions may appear in (5) . They are executed in order. A complete list of actions follow:

EXEC <semantic label>	execute the semantic subprogram, beginning at label <semantic label>.
EXEC id	id is a class name with associated semantic program labels (declared CLASSLAB). It must also appear in (2) of this production. Control is given to the semantic program beginning at label <semantic label> where <semantic label> is the label associated with the reserved word or internal symbol in that stack position.
SCAN	Execute the next phase (which is "on") in this pass with a higher phase number. If the phase uses the production technique, this consists of pushing the symbol in location

SCANSYM (scanned symbol) onto the stack for thst phase where it last executed a SCAN. SCANSYM is pushed onto the stack using the statements PUSH (<main stack identifier>); <main stack identifier> .2 ¬ 1 .= SCANSYM.1; <main stack identifier> .3 ¬ 1 .= SCANSYM.2 (See Section 5 below). If all the phases have been executed using the symbol in SCANSYM, then scan the next input symbol and store it in SCANSYM. Execute the first phase as outlined above.

SCAN	n	SCAN n is equivalent to SCAN ... SCAN (n times).
GO	id	begin the matching at the production labeled id. Any actions following a GO action in a production will never be executed.
CALL	id	execute the productions starting at the one labeled id, and continue until the action RETURN is performed. This may be recursive.
RETURN		return to the production which made the last call.
HALT	n	halt execution of productions, store n in output buffer, return to system control.

ERROR	n	store error message n in output buffer.
UNSTK	m	remove the top m entries from the stack and save in storage (including the semantic portions) $1 \leq m \leq 5$.
STK	n	take the n^{th} entry (counting from the top) removed by the last executed UNSTK action and push it onto the stack (including both the syntax and semantic words).
STAK	id	push id (a reserved word or internal symbol) onto the stack.
STAK	SCANSYM	push the symbol in location SCANSYM onto the stack.
WHEN SIGNAL GO <label>		SIGNAL is local to the phase and has type BYTE (TRUE = $(11111111)_8$ or FALSE = 0). It may be set by the semantic program. If SIGNAL = TRUE, production matching begins immediately at the designated production.

6. Remarks

Three frequently used symbols which are part of the syntax meta-panguage are ANY, I (identifier) and N (number). When used in part ② of a production, ANY matches any item in the stack.

SCANSYM is a system name which is global to all passes and contains the last scanned source program symbol. It is defined as

STRUCTURE T (BYTE2 , BYTE2)

T SCANSYM

When the scanner constructs the next source symbol, it puts its internal representation (if the symbol is not an identifier or number) into SCANSYM.1. If the symbol is an identifier (or number), the representation for "I" (or "N") is put into SCANSYM.1 and the internal representation of the identifier (or number) itself is put into SCANSYM.2. Just before returning to a phase (which has syntax) following a SCAN, the following statements are executed:

PUSH <main stack identifier>

<main stack identifier> .2 ← SCANSYM.1

<main stack identifier> .3 ← SCANSYM.2

References:

[1] Feldman, Jerome A., "A Formal Semantics for Computer Oriented Languages," Thesis, Carnegie Institute of Technology (May 1964).

[2] Gries, D., "The use of transition matrices in compiling," Technical Report CS 57, Computer Science Department, Stanford University, (March 17, 1967).

[3] Mondshein, L. F., "VITAL Compiler-Compiler System Reference Manual," Technical Note 1967-12 Lincoln Laboratory (Feb. 1967).