CGTM #103
Gary Goodman
David Gries
March 1968

COMPILER IMPLEMENTATION SYSTEM

The lexical analyzer definition and the internal dictionary

1. Introduction
2. Syntax
3. Semantics
4. The internal dictionary and statements concerning it
5. Character by character mode

1. Introduction

The <lexical analyzer definition> is used primarily to indicate to the compiler-compiler how to build a lexical analyzer, or scanner. The scanner is that part of a compiler which reads in the original source program characters and composes them into atoms - identifiers, unsigned integers, symbols and reserved words. Each such atom is represented internally by a 16 bit positive integer constant (type BYTE2) which is assigned by the compiler-compiler. Although the compiler writer writes his compiler in terms of the original atoms, the compiler actually works instead with their 16-bit representations. The compiler writer need never know which 16-bit integer represents each atom.

In order to make this mapping from atom to internal representation, each compiler automatically uses an internal dictionary. The compiler writer need not know the details of this dictionary and has little control over it. We mention it here only because it may be used by him to efficiently search his own symbol tables. This is explained later in Section 4.

Besides specifying how atoms are formed, a compiler writer may also test the characters in each input line before it is scanned, and insert other characters into the line at any point. Thus he can test for a FORTRAN "continuation card" or change a fixed field formatted card into a free field line by inserting "break" characters at the end of each field.

A line is limited at all times to a maximum of 250 characters.

We have attempted to define the scanner definition so that the IBM 360 translate and test instructions could be used, along with 3 or 4 256-byte tables. However, we have also tried to define it in such a way that most of the existing source-language representations could be handled. If it

is not possible to build atoms correctly, the compiler writer may at any time switch to a character-by-character mode, in which he receives one character (not an internal representation) at a time from the scanner. He can thus build atoms himself.

This memo also describes various related statements which the compiler writer may use to control the scanning process.

2. Syntax

<lexical analyzer definition> ::= SCAN [synonym] ⊛ [<set name> <char set>] ⊛
[<reserved set>] ⊛ [<string set> / <comment set>] ⊛ [BEGIN <statement> * /] ENDSCAN

<synonym> ::= SYN [<syn identifier> [<char> / <res word>]]*

<syn identifier> ::= <identifier>

<set name> ::= DIGIT | IDBEG | IDCHAR | TERMIN | INVTERMIN | IGNORE

<char set> ::= ALL | [<char> / <letter interval> / <syn identifier>]* | NONE

<char> ::= (an EBCDIC character, except "space")
| SPACE | X' <hex> <hex> '

<letter interval> ::= <letter> THRU <letter>

<letter> ::= 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

<hex> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A |
B | C | D | E | F

<reserved set> ::= RES [<res word>]*

<res word> ::= <syn identifier> | <source identifier> |
<terminator> <terminator> | <terminator>
<source identifier> [<terminator> /]

<terminator> ::= (a member of the set TERMIN)

2. Syntax (continued)

<string set> ::= STRINGQ [<begin symb> <end symb>]*

<comment set> ::= COMMENTQ [<begin symb> <end symb>]*

<begin symb> ::= <res word> | <terminator>

<end symb> ::= (a <res word> consisting of two terminators) |
<terminator>

3. Semantics

a) <synonym>. The <syn identifier> is a synonym for the <char> or <res word> of the source program. This allows one to refer freely to the source program characters and reserved words, especially in syntax subprograms. For example

SYN RBEG BEGIN RPLUS +

indicates that RBEG is a synonym for the source program reserved word BEGIN and that RPLUS means +. The <syn identifier>s follow the usual rules for identifiers in a compiler-compiler program. The <syn identifier>s must appear later in some set definition. The <syn identifier>s, <char>s and <res word>s must be separated from each other by at least one blank space. (This restricts the use of spaces in source programs.) The following words, if reserved words in the source language, must always be represented by a synonym: SYN, ALL, NONE, RES, BEGIN, STRINGQ, COMMENTQ, ENDSCAN, DIGIT, IDBEG, IDCHAR, TERMIN, INVTERMIN, IGNORE, SPACE. It may be advantageous to use synonyms for others — see the restrictions in the syntax subprogram write-up (CGTM 102).

b) <set name> <char set>. This is used to put the characters in <char set> into certain sets. Their meaning is as follows:

1) DIGIT. source integer atoms are formed from members of this set — <source integer> ::= <char defined in DIGIT set>*. The usual characters in this set are 0, 1, ..., 9. When an atom <source integer> is formed, the metasymbol "N" is returned by the scanner, along with the internal number representing this particular string of symbols <source integer>. Note that the number is not converted automatically.

2) IDBEG, IDCHAR. source identifiers are formed using the syntax <source identifier> ::= <char in set IDBEG> <char in set IDCHAR>⊛. When source identifier is recognized by the scanner, it returns the meta-symbol "I", along with the internal number representing the <source identifier>.

3) TERMIN. characters in this set are considered to be single-character atoms of the source program. Examples in ALGOL or FORTRAN: + - , ( ) .

4) INVTERMIN. characters in this set will signal the end of an atom being formed. For example, a blank space following an identifier in some languages "ends" that identifier; A B is two identifiers — A and B. However, these characters are "INVisible" — they are not passed on to the compiler. The end of a line does <u>not</u> signify to the scanner the end of an atom being formed; this may be accomplished by inserting a character from class INVTERMIN at the end of the line.

5) IGNORE. these characters are completely ignored if they appear in the input source program. An example is the blank space in some ALGOL implementations, where A BC is the identifier ABC and 'BE GIN' is <u>begin</u>.

The following restrictions are placed on the sets.

a) The <char> X'70' may not be used; it is reserved for internal use.

b) The sets IDBEG and IDCHAR, or IDCHAR and DIGIT, may have a nonempty intersection; the intersection of any other two sets must be empty.

c) <char set>. ALL means all characters not explicitly declared in other sets, except X'70' and SPACE. NONE means no characters. Any EBCDIC character represents itself (and its 8-bit internal representation). SPACE represents the character whose 8-bit representation is X'40' (on a card this is a blank column). X'<hex> <hex>' represents that internal character. <letter> THRU <letter> represents all those letters — for example, G THRU M represents the letters G,H,I,J,K,L and M. A <syn identifier> represents the character defined as its synonym. They may be used as inserted characters (see below) to end lines or end fixed fields.

d) <reserved set> and <res word>. These are

   1) the usual reserved words in source program — like BEGIN, IF, ELSE, DIMENSION, END.

   2) double-character atoms — like //, /*, */ in PL1.

   3) atoms, such as 'BEGIN', 'END', .EQ., used in some ALGOL implementations.

e) <string set>s and <comment set>s. The <string set> yields sets of beginquotes and endquotes for strings. When a beginquote is detected

in the source program at compile time, a string is formed - of all following characters except ' the internal representation X'70' , and a corresponding endquote - until the following endquote is detected. If the endquote is a symbol consisting of two terminators these terminators must be adjacent - no characters in the set IGNORE will be ignored.

When a string is formed, the symbol "S" (for string) is stored into &SCANSYM.1 , while the string/atom itself is stored into &SCANSYM.2-3

The beginquotes and endquotes in a <comment set> are used in the source program to delimit comments; these are completely ignored.

f) Default options

The following default definitions apply in case a set is not defined.

DIGIT	0 1 2 3 4 5 6 7 8 9
IDBEG	A THRU Z
IDCHAR	A THRU Z 0 1 2 3 4 5 6 7 8 9
TERMIN	NONE
INVTERMIN	SPACE
IGNORE	NONE

g) The <statement>. The <statement> is executed just after a new line of source program has been read in and before it is scanned. The purpose is to allow the compiler-writer to examine and change the line if necessary. This is done using statements explained in CGTM 106. Since this <statement> is not within a certain pass or phase, the

only identifiers it may use are the system identifiers. However, it may call a subroutine of some phase which is executable at this time.

As an example, we give a statement for a FORTRAN continuation card. The FORTRAN card is also changed from fixed to free field format by inserting two terminators (CARDSTEP and LABELEND);

```
IF SUBSTR (εINPUT,6,1) = ' '
THEN εSCANNER .= CARDSTEP CAT SUBSTR(εINPUT,1,5) CAT LABELEND CAT
                        SUBSTR(εINPUT,7,64)
ELSE εSCANNER .= SUBSTR(εINPUT,7,64);
```

If the BEGIN <STATEMENT>* are missing, the following is assumed:

```
BEGIN εSCANNER .= εINPUT;
      εOUTPUT ( εINPUT);  εOUTPUT;
```

4. The internal dictionary and statements concerning it.

All terminators and reserved words are stored in an internal dictionary, along with their internal representations. Identifiers, and numbers and strings are added to the list, at compile time, as they appear in the source program. This dictionary is hash coded for fast access to the entries.

Each entry contains a pointer to the actual symbol, the internal representation and another byte which indicates the type of entry - reserved word or terminator (R), identifier (I) , number (N) , or string (S).

Compilers usually have symbol tables , or dictionaries, consisting in general of one entry per identifier. These may be implemented by TABLES (cf CGTM 101 pp. 16,17,25,26). However, such tables are searched linearly. As an alternative, DICTs have been introduced. They are declared like TABLEs, but with the word DICT instead of TABLE. A further restriction is that the structure definition defining the components must start with

STRUCTURE <structure id> (BYTE, POINTER,

These first two components will be used to chain all entries in DICTs of a single identifier, using the internal dictionary element for that identifier as a base. We give an example after explaining the three basic operations on DICTs.

The operand LOOK:

(1) LOOK (<dict identifier> , <expression>)

(2) LOOK (<dict identifier> , <pointer expression>)

The <expression> must yield a BYTE2 result, which is the internal representation of some atom, or a pointer which is the address of some element in the internal dictionary. The value of the first operand is found by starting at the internal dictionary element for <expression> and searching the chain of elements until one is found in the DICT <dict identifier>.

With the second operand, the <pointer expression> must yield the address of an element in some DICT. The value of the operand is the address of the next element on the chain which is in the DICT <dict identifier> (0 if none).

The operand ENTER:

ENTER (<dict identifier> [,<expression> [,<exp>]])

A new element is added to the DICT <dict identifier> – its value is the value of the <exp> (if there). The <expression> must be either the internal representation of some atom or a pointer to an internal dictionary element. The new element of the DICT is added to the beginning of this atom chain.

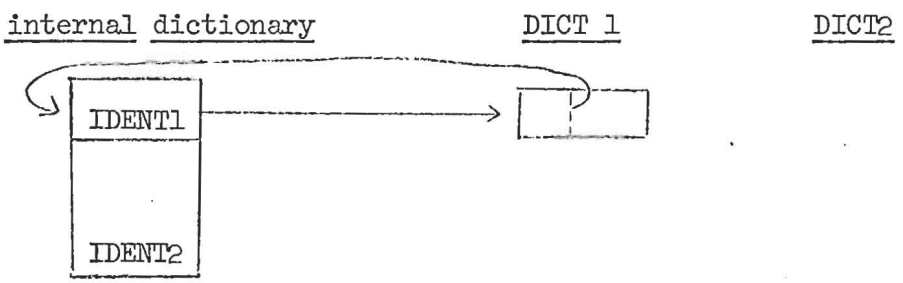
The value of the operand is the address of the new element in the DICT.

The statement DELETE

DELETE (<expression>, <pointer expression>)

The DICT element whose address is <pointer expression> is deleted from the chain on which it appears (but not from the DICT). The appearance of the BYTE2 <expression> indicating the base of the chain is not necessary but is more efficient.

Example: Suppose we have

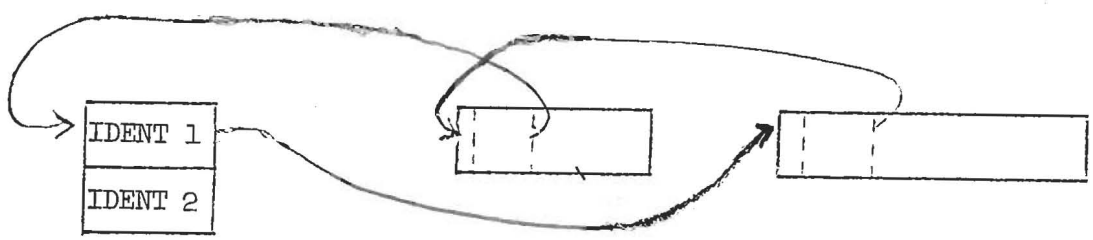


The operations P ← LOOK (DICT2, (internal rep for IDENT1));

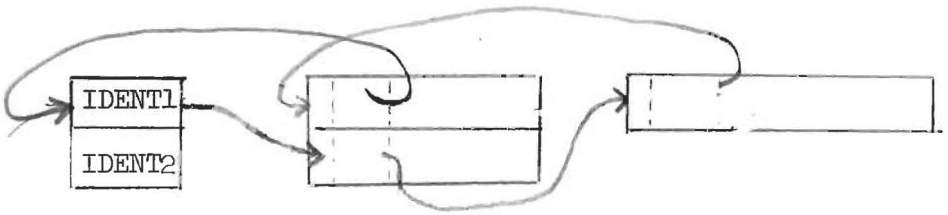
IF P = 0 THEN P1 ← ENTER(DICT2)

ELSE P ← ENTER (DICT1, (int. rep for IDENT1))

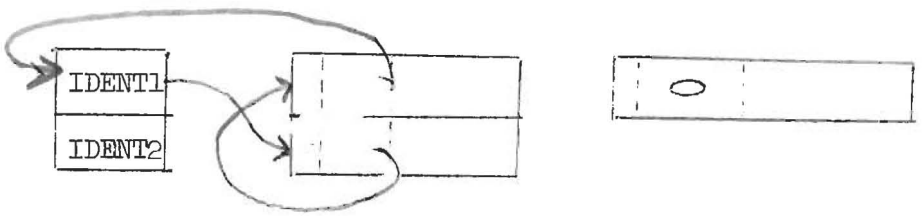
change it to



Executing the same statements again would produce



Execution of DELETE (P1) would yield



In order to give the compiler writer some control over the internal dictionary, the following operands and statements may be used in the semantic sublanguage.

a) Operand

LOOK (INTDIC, <string expression>)

The internal dictionary is searched for an entry equal to the <string expression>. The value of the operand is the address of the entry (0 if not found).

b) ENTER (INTDIC, <string expression> [,S | ,I | ,N])

The value of the <string expression> is entered into the internal dictionary, if not there, with type I, S, or N (default option is I) and assigned an internal number. The value of the operand is the address of the entry.

c) ATOM (<pointer expression>)

The value is the internal representation (BYTE2) of the atom whose element in the internal dictionary is at <pointer expression>

d) ATOM (<string expression>)

Equivalent to ATOM (ENTER (INTDIC, <string expression>))

5. character by character mode.

The execution of the statement

CHARMODE

causes the lexical analyzer to switch to a character-by-character mode. In this mode atoms are not formed. One character is passed to the compiler with each SCAN (the character X'70'is never passed). The internal symbol representing "CHAR" is inserted into &SCANSYM.1, while the character itself is put into &SCANSYM.2 (right adjusted). These are then put into the stack of the phases using a syntax sublanguage.

Execution of the statement

NORMODE

causes the scanner to resume forming atoms.

In order to aid in forming strings when in character by character mode the following string operand may be used in the semantic sublanguage:

CAT (<pointer expression>$_1$ [,<pointer expression>])

Both <pointer expression>s must be the address of some element of the main stack in which the operand is used; the default option for the second is the top element. The value is the string formed by the characters in <pointer expression>$_1$.2, ... , <pointer expression>$_2$.2 .