CGTM #101
David Gries
November, 1967

Compiler Implementation System

This second memo describes the structure of a compiler written in the language, and the ALGOL-like facilities in the semantic sublanguage. Changes will definitely occur, and some vague points will be clarified as the implementation progresses.

5. Expressions

    5.1 operands

    5.2 expressions

    5.3 structure expressions

    5.4 component table expressions

6. Statements

    6.1 assignment statements

    6.2 iterative statements

    6.3 control statements

    6.4 case statements

    6.5 phase and pass communication statements

    6.6 stack and table statements

    6.7 other statements

7. Alignment

1. Introduction

1.1 Overview

This note describes the overall structure of the language in which a compiler is written and gives part of the definition of the semantic sublanguage. Other notes will contain discussions and complete descriptions of specific parts of the language, as follows:

CGTM 100	The general discussion of compilers written in this language.
CGTM 101	This report - overall structure and the semantic sub-language.
CGTM 102	The syntax sublanguage.
CGTM 103	The lexical analyzer definition and the internal dictionary.
CGTM 104	The DESCRIPTOR and operations on it.
CGTM 105	Code brackets and statements for controlling code generation.
CGTM 106	input-output statements

We assume that the reader is familiar with CGTM 100.

Although one would like a completely machine independent language for writing compilers, this is not feasible. For example, descriptions of run-time operands (variables, constants, etc.) at compile time must necessarily be in terms of the machine for which code is to be generated. However, we try to make the language as machine independent as possible. This note does not describe <u>all</u> possible constructs in the semantic sublanguage - just those which are "ALGOL-like". Others will be described in CGTM 102, 103, 104, 105, 106.

The semantic sublanguage is an ALGOL-like language with special facilities added:

(1) the ability to construct new data types as a composition of other previously defined types;

(2) tables, stacks and pointer variables;

(3) the ability to construct new data elements dynamically without having to assign them to predeclared variables, and the ability to "free" the space allocated to such data elements;

(4) the concept of "code brackets" (FELDMAN [1]); operators and operands in code brackets cause code to be <u>generated</u>, not executed;

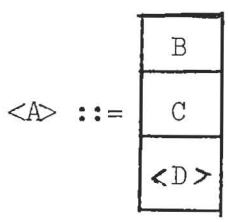
(5) statements to control and describe how code is to be generated;

(6) control statements to link syntax with semantics, to communicate between "phases", and to link passes together.

1.2 Conventions used in the description of the language

The syntax is described in a modified version of BNF using some of the ideas of Dan Ross (Box Syntax, SLAC, CGTM No. 16) in order to provide a more readable syntax description. Nonterminal symbols are always enclosed in angular brackets "<" and ">".

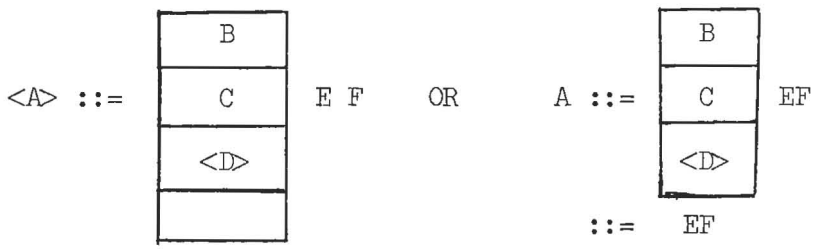
Besides the symbol "|" to denote alternates as in BNF, a rectangular box may be segmented vertically by horizontal lines, with each alternate put into one box. Thus <A> ::= B | C | <D> may be equivalently written as



This allows us to write

<A> ::= B E F | C E F | <D> E F | E F

in the more readable form:

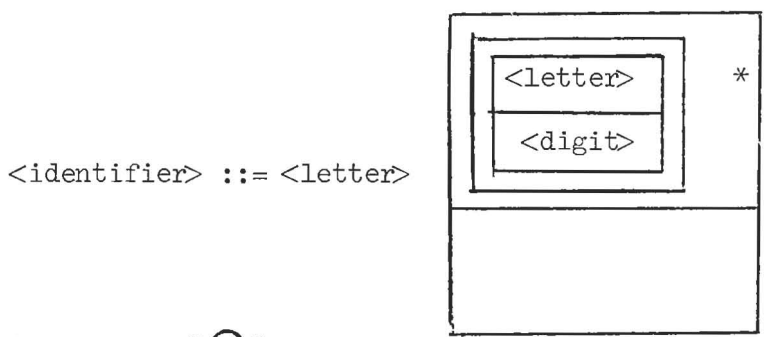


Note the use of an empty box to denote the empty string.

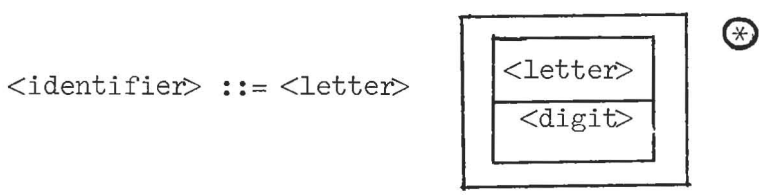
Several constructs may be enclosed in a single box. An asterisk "*" following a box means 1 or more repetitions of the constructs in the box. Thus

<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>

may be written as



With the use of "⊛" to represent zero or more repetitions, we write this as



or as

<identifier> ::= <letter> <alphanum> ⊛

<alphanum> <letter> | <digit>

A final modification lets one easily describe a list of objects (say identifiers), separated by some punctuation. In this case, the object is put in a box, a dotted box containing the punctuation is adjoined on the right and the repetition symbols "*" or " ⊛ " are used. Thus

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

can be written

<id list> ::= $\boxed{\text{<identifier>}}$ $\boxed{,}$ *

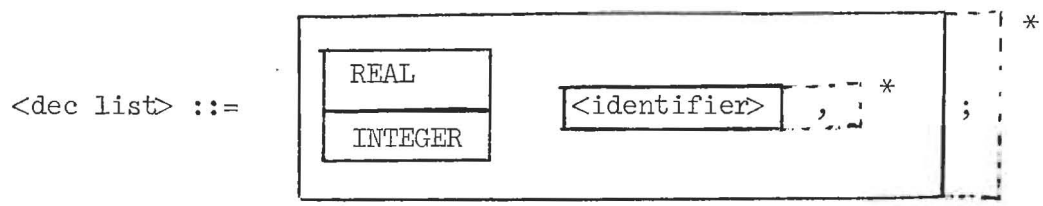
A more complicated example is

<dec list> ::= <declaration> | <dec list> ; <declaration>

<declaration> ::= REAL <id list> | INTEGER <id list>

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

which could be written as



Occasionally, a comment in parentheses will appear instead of the right had part of the production.

1.3 Code brackets

Code brackets "CODE(" and ")" are used to indicate that code is to be generated for the object program. In general, an expression or statement appearing within code brackets causes code for that expression or statement to be generated, not executed. Code brackets are an important concept in compiler writing. They free the compiler writer from having to worry about the grubby details of code generation (such as register allocation); he is writing his compiler on a much higher level. Of course, constructs are provided to give the experienced compiler writer as much control over code generation as he wants.

Basic to the concept of code brackets is a data type called DESCRIPTOR (or DESCR for short). DESCR, a composition of several other basic data elements, is used to describe runtime operands, so that code may be generated properly. The DESCR has one component for describing the "kind" of operand (floating point, integer, pointer, etc.), another for describing its "size" (1, 2, 4 bytes, etc.), another its location, etc.

All the code generation routines rely on elements of type DESCRIPTOR for operands.

For instance, if the statement

CϕDE (D1 + D2)

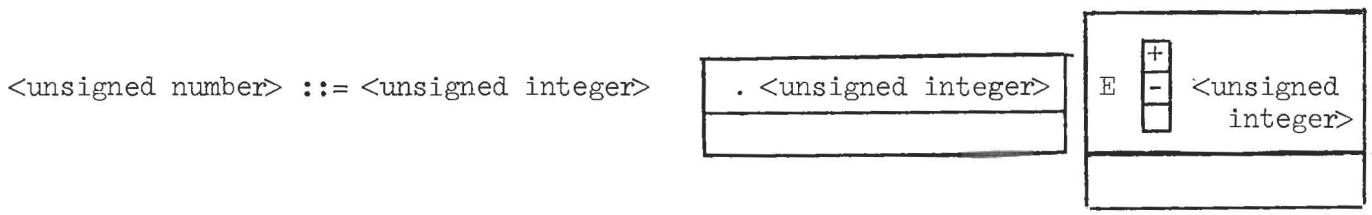
is executed, code is generated to add the values of runtime variables (say A and B) described by the DESCRIPTOR variables D1 and D2. A new DESCRIPTOR element is then generated to describe the result of the expression A + B at runtime (its kind, size, location, whether in a register or not, etc.).

In general, the only operands that can appear within code brackets are constants, variables declared in the compiler to be valid at runtime, or compile time variables whose type is DESCRIPTOR (which describe actual runtime variables).

1.4 Basic symbols, identifiers and numbers

<letter>	::=	(the letters A through Z)
<digit>	::=	(digits 0 through 9)
<delimiter>	::=	(list will be given later; contains <basic symbols>, +, -, etc., and <reserved word>s BEGIN, END, IF, etc.)
<identifier>	::=	<letter> [<letter> / <digit>] ⊛
<unsigned integer>	::=	[<digit>] *

Blank spaces may not appear between characters of <basic symbol>s, <reserved word>s, <identifier>s or <unsigned integer>s; at least one blank space must separate adjacent <identifier>s, <reserved word>s or <unsigned integer>s. Other than that, blanks have no meaning except in strings and may be used freely. The end of a card has no particular significance, as in ALGOL. Certain <reserved word>s may not be used as <identifier>s; this list will be given later. Identifiers may designate variables, structure types, components of structures or may be used as labels in a program.



<unsigned number>s may be fixed-point (integer), in which case the fraction and exponent part must be omitted, or floating-point (real). The exponent part specifies a power of ten.

2. Compiler structure

2.1 Syntax

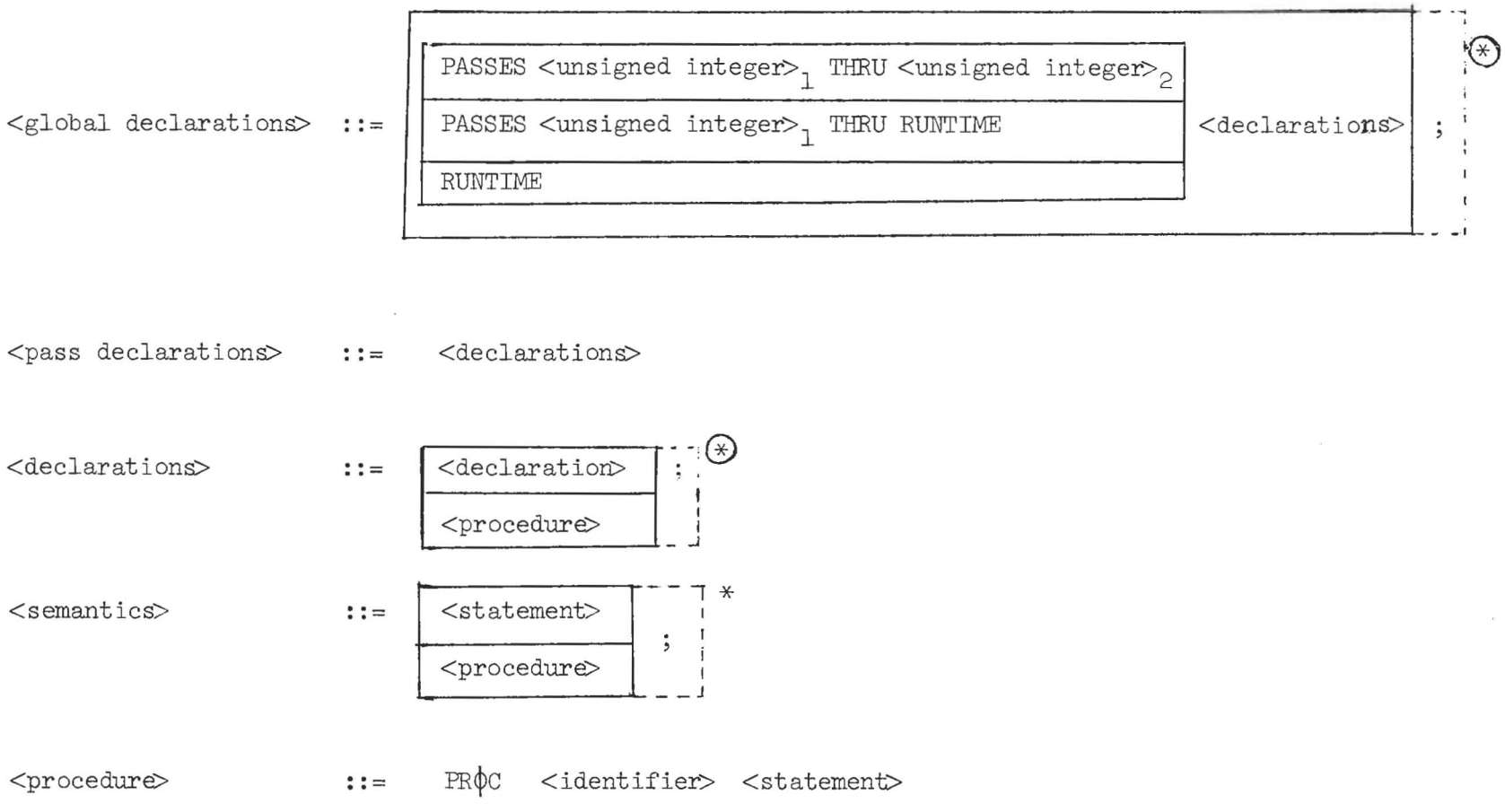
<compiler> ::= BEGIN <lexical analyzer definition><global declarations> <passes> END

<passes> ::= <pass> ; *

<pass> ::= PASS <unsigned integer> <pass declarations> <phases> ENDPASS

<phases> ::= <phase> ; *

<phase> ::= PHASE <unsigned integer> < declarations> <syntax> <semantics> ENDPHASE


<global declarations> ::=
PASSES <unsigned integer>1 THRU <unsigned integer>2
PASSES <unsigned integer>1 THRU RUNTIME
RUNTIME
<declarations>
;
*
<pass declarations> ::= <declarations>
<declarations> ::=
<declaration>
<procedure>
;
*
<semantics> ::=
<statement>
<procedure>
;
*
<procedure> ::= PRØC <identifier> <statement>

2.2 Semantics

The <lexical analyzer definition>defines the basic symbols, reserved words, etc. which can appear in a source program to be compiled at compile-time. The metacompiler will build a scanner from these definitions. When asked for "the next input symbol" during compile time, this scanner will give to the compiler a representation of the next reserved word, basic symbol, integer, or number in the source program, thus freeing the compiler writer from having to build a scanner. This <lexical analyzer definition> is described in CGTM 103. Internally, each reserved word, basic symbol, integer or number is represented by a (16 bit) positive number.

<global declarations> define data elements, and their identifiers, which are to be used by more than one pass of the compiler and/or at runtime. Identifiers declared in <global declarations> are valid in the pass numbered <unsigned integer>$_1$, through the pass numbered <unsigned integer>$_2$ (or through runtime), or just at runtime, depending on the construct used.

Passes are executed serially, in the order defined by the <unsigned integer> following the symbol PASS. Each pass may be thought of as a "coreload". <pass declarations> define data elements and their identifiers which are to be used in a particular pass. If an identifier is declared both in <global declaration> and a <pass declaration>, its use within the pass always refers to its definition in the <pass declaration> (the usual block structure concept).

Each pass may consist of a number of phases. These are executed, as described in CGTM 100, in the order given by the <unsigned integer>

following the symbol PHASE. Each phase may or may not have a <syntax> subprogram, but must have a <semantics> subprogram. This present report is mostly concerned with the <semantics> subprogram sublanguage.

<declarations> within the phase define data elements and their identifiers which are to be used only in the particular phase. The use of an identifier declared both within a phase and in a <global declaration> or <pass declaration> refers to its declaration in the phase (usual block structure concept).

When a pass is finished, the identifiers, and their values, declared within the <pass declaration> and within its phases lose their validity. This occurs even when a "pointer" variable declared in a <global declaration> points at an identifier declared within the phase. The metacompiler cannot check this.

The <semantics> is a sequence of statements or procedures. The syntax subprogram may execute the semantics, beginning at any labeled statement <u>which is not part of a procedure</u>. The <procedure>s can be placed anywhere in the sequence, but are not executed until called.

As can be seen, procedures written by the compiler writer may not have formal parameters. This may be changed later if the need arises.

3. Declarations

3.1 <type>s

3.1.1 syntax

<simple type> ::= <arith type> | <bits type> | <pointer type> | <string type>

<type> ::= <simple type> | <structure type>

<arith type> ::= FWF | DWF | FWI | HWI | DEC

<bits type> ::= BYTE | BYTE2 | BYTE3 | BYTE4

<pointer type> ::= PØINTER [(TØ <pointer list>)]

<structure type> ::= <structure id>

<pointer list> ::= [<pointer to>]*

<pointer to> ::= TABLE | STACK | <type> | <table identifier> | <stack identifier>

<string type> ::= STRING

3.1.2 semantics

<type>s are used later in declarations to define properties of quantities associated with identifiers (or variables). FWF, DWF, FWI, HWI and DEC denote Full Word Floating point, Double Word Floating point, Full Word Integer, Half Word Integer, and DECimal respectively. BYTE, BYTE2, BYTE3, BYTE4 specify positive integers (or strings of bits) of 8, 16, 24 and 32 bits respectively.

POINTER specifies a pointer variable which may hold the address of (a reference to) a variable of any <type>, an element of a TABLE or STACK (see below), or to a TABLE or STACK itself. A reference to a pointer variable must usually be qualified to indicate the <type> of the data element to which it refers. This may be indicated in the <type> itself by using the form

POINTER (TO <pointer list>)

A pointer variable declared with this <type> is then restricted to point to only the <type>s of data elements defined by the <pointer list>, as indicated in the following table.

<pointer to> component	the pointer may refer to
TABLE	any TABLE as an entity
STACK	any STACK as an entity
<type>	a data element of this <type>
<table identifier>	an element of the designated TABLE
<stack identifier>	an element of the designated STACK

STRING represents a string of from 0 to 255 characters.

A <structure type> is a <type> defined by the programmer in a <structure definition> (see Section 3.2).

3.2 Structure definitions

3.2.1 syntax

<structure definition> ::= STRUCTURE <structure id> <component descriptions>

<structure id> ::= <identifier>

<component descriptions> ::= (<component description> , *)

<component description> ::= <component descriptions> | <component identifier> <component descriptions> | <simple component> ALT *

<component identifier> ::= <identifier>

<simple component> ::= <type> | <component identifier> |

3.2.2 semantics

<structure definition>s are used to define new <type>s of data elements by describing a composition of already-existing <type>s. A <structure type> consists of one or more <component>s, separated by

commas. These <componet>s may be named or unnamed, and may themselves contain subcomponents. <component>s are numbered (by the metacompiler), from left to right, starting at 1.

A <component> may have several alternatives, specified by "ALT". This concept is introduced mainly to save space in memory. Only one of the alternates may be in existence at any time, and it is up to the programmer to know which one is being used. Alternates for a <component> are numbered (by the metacompiler) from left to right, starting with 1.

It is not necessary for the compiler writer to know how the different components of a variable of some structure type are aligned in memory. However, this knowledge is useful because one may save space by properly sequencing the components. See Section 7.

If a <structure type> is used as the type of some <component identifier>, this <structure type> must have been previously (statically) declared.

3.2.3 examples

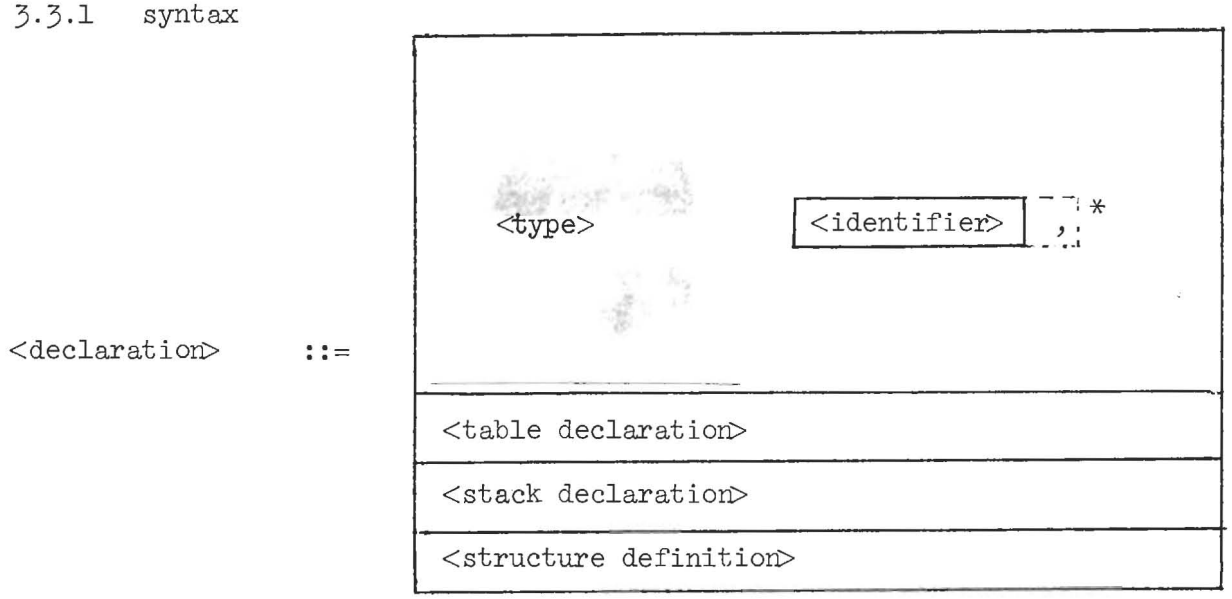
```
1.  STRUCTURE SUBSCR   (  BYTE AREA,
                          BYTE3 OFFSET1,
                          POINTER SUBDESCR)

2.  STRUCTURE DESC     (  HWI KIND,
                          HWI USE,
                          (BYTE AREA, BYTE3 OFFSET)
                       ALT  POINTER (TO SUBSCR) P ,
                          SUBSCR A,
                          SUBSCR B    )
```

Example 1 illustrates a <structure type> SUBSCR which consists of three components; the first is 8 bits in length and named AREA, the second is 24 bits long and named OFFSET. The third is a pointer named SUBSDESCR.

Example 2 defines a <structure type>, DESC, consisting of five components. The first two are half-word integers and named KIND and USE. The third component may take two forms. The first alternate is unnamed and has itself two components — AREA and OFFSET. The second alternate is a pointer variable, P, to data elements of <type> SUBSCR. The programmer must keep track of which alternate is being used. The last two components are of type SUBSCR.

3.3 <declaration>s

3.3.1 syntax



<table declaration> ::= [STATIC | DYNAMIC |] <type> TABLE (<unsigned integer>) <table identifier>

<table declaration> ::= [STATIC | DYNAMIC |] <type> DICT (<unsigned integer>) <table identifier>

<table identifier> ::= <identifier>

<stack declaration> ::=
- [STATIC | DYNAMIC |] <type> STACK (<unsigned integer>) <stack identifier>
- MAIN <type> STACK (<unsigned integer>) <stack identifier>
- MAIN STACK <stack identifier>

<stack identifer> ::= <identifier>

3.3.2 semantics

As in ALGOL, declarations serve to declare identifiers and the type of data which can be assigned to them. A <table declaration> describes a table of elements of a specified <type> (which may of course be some structure type). A DICT is the same as a TABLE, except for the way certain ~~options~~ operations are performed on it (searching and entry). See CGTM 103 for a complete description and restrictions on DICT.

A table may be declared STATIC (in which case <unsigned integer> gives the number of elements in the table) or DYNAMIC. In the latter case, the <unsigned integer> defines the number of elements in a "block" of the table; space will be initially allocated to one block; and extra blocks will be added at compile time or runtime as the need arises. The default option is DYNAMIC.

A <stack declaration> defines a pushdown stack (LIFO). STATIC, DYNAMIC, and <unsigned integer> have the same meaning as in the table declarations. Each phase which uses production language must have a MAIN STACK declared in it; such a stack must be STATIC (for efficiency purposes). CGTM 102 describes in detail certain restrictions on MAIN STACKs.

Suppose a stack had been declared in a <global declaration> or <pass declaration> by

STATIC <type> STACK (100) S1

and this stack is to be the main stack of a phase. Then, in that phase, the following declaration should be inserted:

MAIN STACK S1.

See CGTM 102 for a complete description.

3.3.3 examples

```
FWF A1, A2

BYTE D1

SUBSCR B1, B2, B3 (See examples in Section 3.2.3)

FWF TABLE (100) C1

MAIN  DESC  STACK (500)  E1

PØINTER  (TØ FWF) P,

PØINTER  (TØ SUBSCR)  P2

PØINTER  P1, P3
```

4. Referencing Data

4.1 Syntax

<data ref>	::=	<stack reference>
		<table reference>
		<simple variable reference>
		<component reference>
		<indirect reference>

<data reference> ::= <address reference> | <data ref>

<address reference> ::= @ <addressable operand>

<addressable operand> ::= <data ref> | <label>

<stack reference> ::= <stack identifier> (<arith expression>) | ( )

::= L0 | L1 | L2 | L3 | L4 | L5 | L6 | R0 | R1 | R2 | R3 | R4

<table reference> ::= <table identifier> (<arith expression>)

<simple variable reference> ::= <identifier>

<component reference> ::=

- <stack reference>
- <table reference>
- <indirect reference>
- <identifier>

. <component selector>

<component selector> ::=

- <unsigned integer> [¬ <unsigned integer>] *
- <component identifier> [.] *

<indirect reference> ::= < <data reference> [<type>] >

4.2 Semantics

These constructs are used to access information.

A <stack reference> is used to reference elements on a stack. The <arith expression> is evaluated and truncated to an integer, i (i must be ≥ 0), and the ith stack element down from the top is selected, the top element being the 0th element. If (<arith expression>) is missing, the top stack element is referenced.

If a phase has a syntax subprogram, L0, L1, L2, L3, ... , L6 refer to the top element, first from the top, ..., 6th from the top element, of the main stack of the phase, before the current production was matched. R0, R1, ... , R4 refer to the top element, etc., of the main stack after the production was matched and changed but before any actions were executed. See Section 6.6 and CGTM 102 .

A <table reference> is used to reference TABLE elements. The <arith expression> is evaluated and rounded to the nearest integer, i ($i > 0$), and the ith TABLE element is selected. It is recommended that TABLE elements be referenced using POINTER variables, since this is much more efficient(especially for dynamic tables).

A <simple variable reference> is a symbolic reference to a data element declared with type <arith type>, <bits type>, <pointer type>, <structure type> or <string type>.

A <component reference> is used to reference a particular component or subcomponent of a data element which is a structure. The <...reference> or <identifier> determines a structured data element from which the component is to be selected, while the particular component is described by the <component selector>.

If the first option for the <component selector> is used, the first <unsigned integer> specifies the component of structure. The "¬ <unsigned integer>" specifies the alternate, the default option being "¬ 1". Subcomponents of this particular component are specified in the same way.

If the second option is used, the <component identifier>s, read from left to right, are used to reference the component, subcomponent of the component, subcomponent of the subcomponent,etc. Only enough <component identifier>s need to be present to unambiguously determine the component to be used. See examples in 4.3.

An indirect reference allows indirect access to data elements. The <data reference> must yield an operand of type <pointer type>; the actual data referenced is found at the address given by the value of the <data reference>. It is necessary to indicate, by <type>, the type of the actual data element. If absent, the metacompiler will attempt to determine at metacompile time the type of the actual data element. If not possible, an error message will be given.

An <address reference> allows the address of a data element or a label

to be accessed. The <address reference> yields a result of type <pointer type> which is the address of the leftmost byte to which the <addressable operand> refers. One can reference the address of any data element except another <address reference>.

4.3 Examples

Consider the examples in 3.2.3 and 3.3.3.

E1	references the top element of stack E1.
E1(2)	references the second from the top element.
C1(5)	references the fifth element of table C1.
A1	refers to the variable A1.
B1	refers to the data element (<type> SUBSCR) assigned to B1.
B1.1	refers to its first component (equivalent to B1 AREA).
E1(2). OFFSET1	is ambiguous (equivalent to either E1(2).4.2 or E1(2).5.2.
E1(2).A.AREA	refers to the component named AREA of the structure data element A which is the fourth component of the 2nd from the top element of the stack E1.
<E1.P> or <E1.3 ¬ 2>	refers to the data element of type SUBSCR pointed at by the component named P of the top element on stack E1.

@E1	refers to the address of the top element of stack E1.
<@E1.P>	is the same as E1.P.

5. Expressions

5.1 Operands

<operand> ::= <data reference> | <function designator>
| <code statement> | <constant> | <table pointer> |
<data reference> (<expression>$_1$: <expression>$_2$)

<function designator> ::= <function identifier> ([<paramter>] , *)

<function identifier> ::= <identifier>

<parameter> ::= (to be explained later)

<code statement> ::= CODE (<runtime state>)

<runtime state> ::= (discussed in CGTM 105)

<constant> ::= <unsigned integer> | <unsigned number>
| <string constant> | $ <source language word>
| <internal identifier> | TRUE | FALSE | NULL

<string constant> ::= #

[<char>] *
$ 2 [<bit>] *
$ 3 [<oct>] *
$ 4 [<hex>] *

#

<char> ::= <any character except $ or #> | $$ | $#

<bit> ::= 0 | 1

<oct> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

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

<table pointer> ::= LØØK (<table identifier> [,<component selector>] ,<expression> [,<pointer expression>] [,BACK | ,FORWARD])

::= ENTER (<table identifier> , [<exp>] , [,BACK | ,FORWARD])

$$ and $# represent the characters $ and # respectively.

The type of each operand is given in the following table.

<operand>	its type (and value, where not obvious)
<data reference>	the type of the <data reference>
<function designator>	the type of the <function designator>
<unsigned integer>	HWI, if <unsigned integer> < 2^{15}-1 else FWI
<data reference> (<exp> : <exp>)	BYTE, BYTE2, BYTE3, BYTE4 (see c)
<unsigned number>	FWF, DWF or DEC, depending on its size and the context
$ <source language reserved word>	BYTE2 (See a) below)
<internal identifier>	BYTE2 (See b) below)
<code statement>	POINTER (to a DESCRIPTOR) see CGTM 104,105
# <char> * #	STRING
# $ 2 <bit> * #	BYTE, BYTE2, BYTE3 or BYTE4.
# $ 3 <oct> * #	BYTE, BYTE2, BYTE3, or BYTE4.
# $ 4 <hex> * #	BYTE, BYTE2, BYTE3 or BYTE4.
TRUE, FALSE	BYTE - $(11111111)_2$ = TRUE, $(00000000)_2$ = FALSE
NULL	POINTER
<table pointer>	POINTER (see below)

a) A <source language reserved word> is a sequence of characters declared in the <source language description> as a reserved word. See CGTM 103. Its value is the 16 bit positive integer assigned by the metacompiler.

b) A <internal identifier> is a name declared as "INT". The value of it is the 16 bit positive integer assigned by the metacompiler(See CGTM 102,103)

c) The <data reference> must have type <simple type>; $1 \leq$ <expression>$_1$ $\leq$ <expression>$_2 \leq 32$. The value is the Bits <expression>$_1$ to <expression>$_2$, right adjusted, with enough leading zeroes to form the smallest of BYTE, BYTE2, BYTE3 or BYTE4.

A <table pointer> yields a value of type POINTER, either NULL or a pointer to some element of the TABLE or DICT <table identifier>. The default option for FORWARD or BACK is FORWARD. If <pointer expression> is missing NULL (0) is assumed. For TABLES, the value of LOOK and ENTER are found as follows:

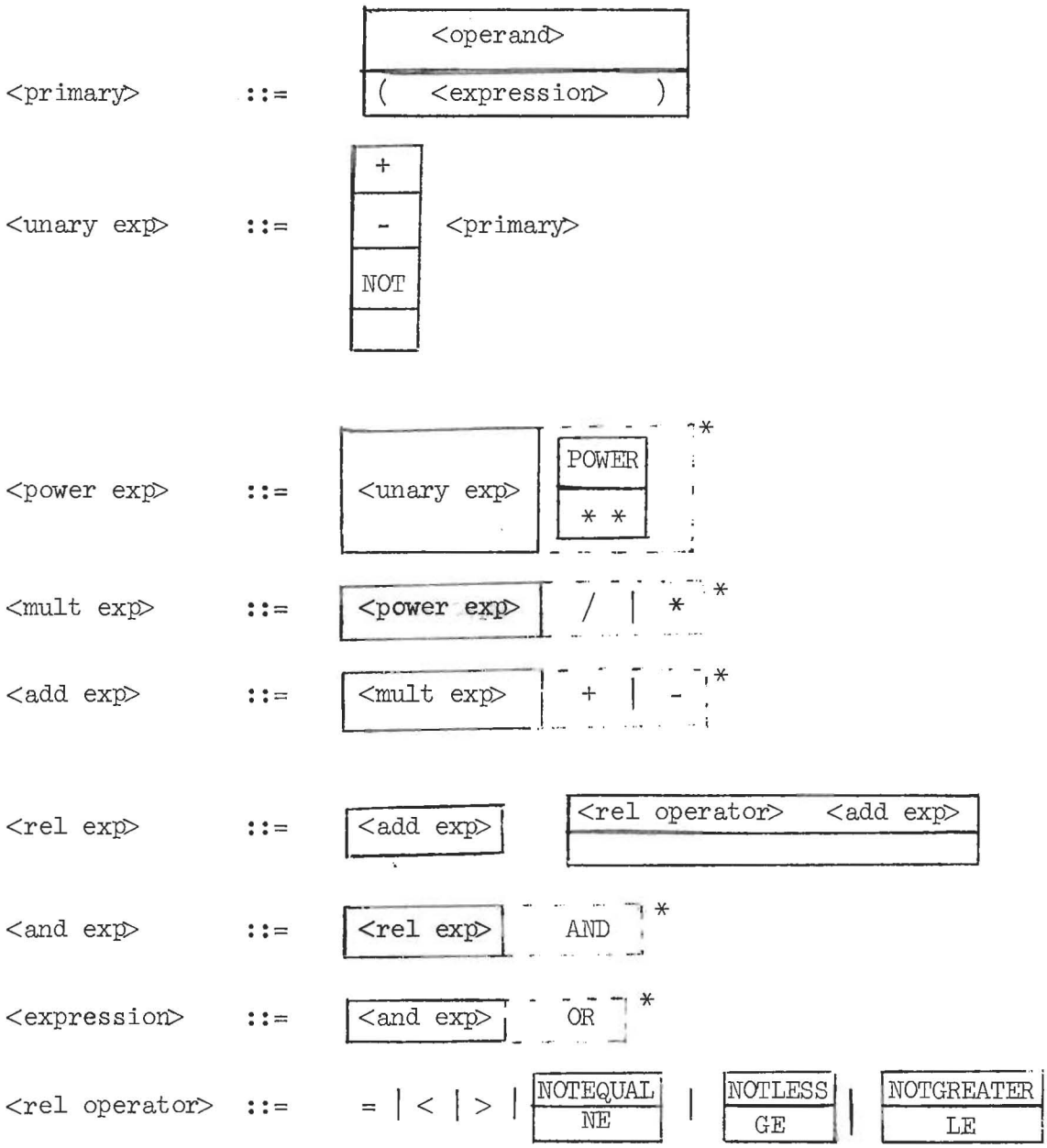
LØØK (FORWARD). Suppose <pointer expression> points at the ith element. The elements i + 1, i + 2, i + 3, ... are tested, in order, for one whose (sub)component defined by <component selector> is equal to <expression>. If one is found the value of LOOK is the address of that element; if not, the value is NULL.

(BACK). As with FORWARD, but the elements are tested in order i - 1, i - 2, i - 3, ..., 1. If <pointer expression> is NULL, the list is searched starting with the highest numbered element.

ENTER A new element is added at the end of the table; its value is the value of <exp> if present; otherwise its value is undefined. FORWARD and BACK have no meaning here. The value of ENTER is the address of the new element.

LØØK and ENTER for DICTS are defined in CGTM 103. They are essentially the same, but a different searching technique is used.

5.2 Expressions



If two operands which do not have the same type are used in some operation, one is converted to the other form. The following table indicates this, besides showing which operands are valid for each operation. With arithmetic operations, an operand of the type <bits> is always assumed to be a positive integer. When an operand of type BYTE4 must be converted to FWI, the leftmost bit is dropped from the value (made zero), yielding a 31 bit positive integer.

Operand	Result of unary + (no operation)	Result of unary -	Result of unary NOT (complement)
BYTE	BYTE	HWI	BYTE
BYTE2,BYTE3,BYTE4	BYTE2,BYTE3,BYTE4	FWI [redacted]	BYTE2,BYTE3,BYTE4
HWI,FWI	HWI,FWI	HWI,FWI	not allowed
FWF,DWF,DEC	FWF,DWF,DEC	FWF,DWF,DEC	not allowed
POINTER	POINTER	not allowed	not allowed

Power (**)

a \ b	BYTE,BYTE2,3,4	HWI	FWI	FWF	DWF	DEC
BYTE,BYTE2,3,4 (see a)	FWI	FWF(see a)	FWF(see a)	FWF	DWF	DEC
HWI,FWI	FWI	FWF(see a)	FWF(see a)	FWF	DWF	DEC
FWF	FWF	FWF	FWF	FWF	DWF	DEC
DWF	DWF	DWF	DWF	DWF	DWF	DEC
DEC	DEC	DEC	DEC	DEC	DEC	DEC

a) if b is an <unsigned integer>, the result will be FWI.

a / b, a * b (if two results are given, the first is for /, the second for *.)

a \ b	BYTE	BYTE2, 3, 4	HWI	FWI	FWF	DWF	DEC
BYTE	FWF,HWI	FWF,FWI	FWF,HWI	FWF,FWI	FWF	DWF	DEC
BYTE2, 3, 4	FWF,FWI	FWF,FWI	FWF,FWI	FWF,FWI	FWF	DWF	DEC
HWI	FWF,HWI	FWF,FWI	FWF,HWI	FWF,FWI	FWF	DWF	DEC
FWI	FWF,FWI	FWF,FWI	FWF,FWI	FWF,FWI	FWF	DWF	DEC
FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC
DWF	DWF	DWF	DWF	DWF	DWF	DWF	DEC
DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC

a + b, a - b

a \ b	BYTE	BYTE2, 3, 4	HWI	FWI	FWF	DWF	DEC	POINTER
BYTE	HWI	FWI	HWI	FWI	FWF	DWF	DEC	POINTER (see c,b)
BYTE2, 3, 4	FWI	FWI	FWI	FWI	FWF	DWF	DEC	POINTER (see b,c)
HWI	HWI	FWI	HWI	FWI	FWF	DWF	DEC	POINTER (see b,c)
FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	POINTER (see b,c)
FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
DWF	DWF	DWF	DWF	DWF	DWF	DWF	DEC	
DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	
POINTER	POINTER (see c)	POINTER (see c)	POINTER (see c)	POINTER (see c)				FWI

b) a - b where b is of type POINTER and a is not, is not allowed.

c) undefined if b = NULL or a = NULL.

a<rel operator>b — Equivalent to c ← a - b, checking whether c is positive, negative or zero as necessary and producing TRUE or FALSE accordingly (i.e., result is BYTE).

a AND b, a OR b — (if a conversion is necessary, zeroes are added to the left of the shorter of a and b).

a = BYTE, BYTE2, 3, 4

b = BYTE, BYTE2, 3, 4

a [AND / OR] b yields type BYTE, BYTE2, 3, 4

5.3 Structure expressions

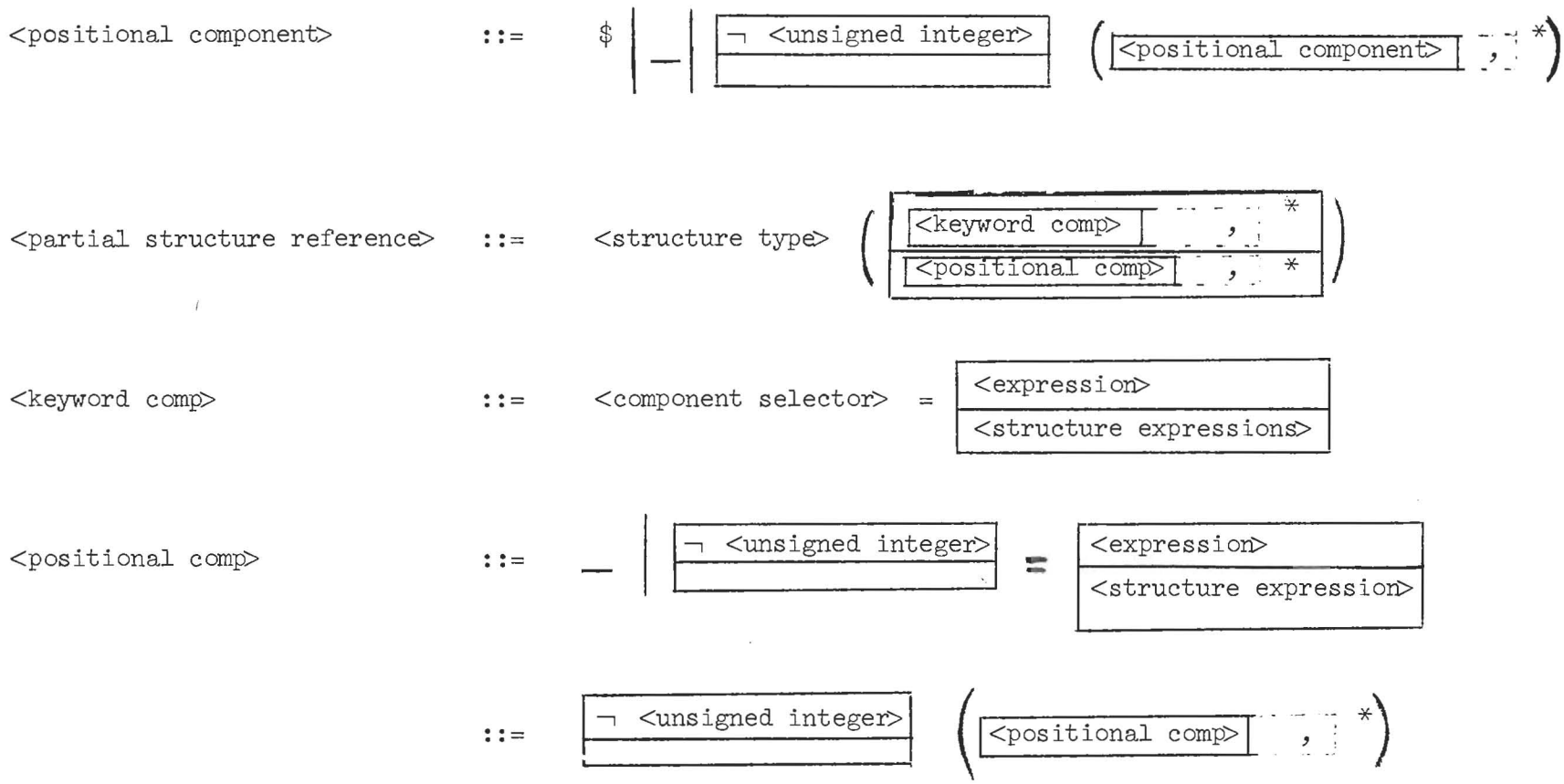
<exp> ::= <expression> | <structure expression>

<structure expression> ::= <complete data reference> | <partial data reference> | <partial structure reference> REPLACE *

<complete data reference> ::= <data reference> | <function designator>

<partial data reference> ::= <data reference> | <function designator> (<keyword component> , * | <positional component> , *)

<keyword component> ::= <component selector>


<positional component> ::= $ | — | ¬ <unsigned integer> (<positional component> , *)
<partial structure reference> ::= <structure type> (<keyword comp> , * | <positional comp> , *)
<keyword comp> ::= <component selector> = <expression> | <structure expressions>
<positional comp> ::= — | ¬ <unsigned integer> = <expression> | <structure expression>
::= ¬ <unsigned integer> (<positional comp> , *)

A <structure expression> yields a data element of some <structure type>. The evaluation is defined as follows:

1. The <structure expression> is a <complete data reference>. Its value is the value of the <complete data reference>.

2. The <structure expression> is a <partial data reference>. Space is allocated for the new structure. The value is the value of the <function designator> or <data reference>, with some of the components set to <u>empty</u>. This is indicated by the <keyword component>s or <positional component>s as follows:

   a) <keyword component>s: Those components or subcomponents named by the <component selector>s are left alone; those not mentioned are set to <u>empty</u>.

   b) <positional component>s: The form is isomorphic with the form of the structure. "$" means the (sub)component is to be used, "_" means to set to <u>empty</u>. When subcomponents are also indicated, ¬ <unsigned integer> specifies the alternate for the component (1 is the default option).

3. The <structure expression> is a <partial structure reference>. Space is allocated for the new structure. The <structure type> defines the type of the <structure expression>.

   a) When <keyword comp>s are used, those components <u>not</u> specified by a <component selector> are set to <u>empty</u>; the others are given the designated <expression> or <structure expression>.

b) When <positional comp>s are used, the form must be isomorphic to the structure. "_" means <u>empty</u>. In the second alternate, " ¬ <unsigned integer>" specifies the alternate for the component (1 is the default option), while <expression> or <structure expression> gives its value.

4. Form A_1 REPLACE A_2 REPLACE ... REPLACE A_n.

A_1, A_2, ... , A_n must be of the same <structure type>. <u>Space</u>, say D, <u>is allocated</u> for a new structure of that type. A_1 is evaluated and stored in the new space. Those (sub) components of A_2 which are not <u>empty</u> replace the corresponding (sub)components in D. This is repeated for A_3, ... , A_n in that order. The result is then in D.

It should be noted that, for all except a <structure expression> which is a simple <data reference>. space is allocated for a new structure. In certain instances the compiler writer is responsible for freeing this storage area when no longer needed. This will be explained later. In all cases where some <expression> is assigned to a (sub)component of a <structure expression>, the assignment is made following the conversion rules given in Section 6.1.

<u>Examples</u> Consider the examples in 3.2.3 and 4.1.3.

Suppose SUBSCR A1, A2 and DESC B1, B2 are two declarations, and suppose

A1 is (1, 20, NULL) A2 is (2, 25, NULL)

B1 is (3, 4, (5, 21), (6, 22, NULL), (7, 23, NULL))

B2 is (8, 9, (10, 24), (11, 25, NULL), (12, 26, NULL))

Then

A1	is a <structure expression> whose value is the value of A1.
A1 ($,_,$)	is a <structure expression> whose value is (1, <u>empty</u>, NULL)
B1 ($,_,($,_),$,_)	has the value (3, <u>empty</u>, (5, <u>empty</u>), (6, 22, NULL), <u>empty</u>)
B1 (1, A, B. AREA, 3.1)	has the value (3, <u>empty</u>, (5, <u>empty</u>), (6, 22, NULL), (7, <u>empty</u>, <u>empty</u>)
B1 ($, $, ¬ 2, _, _)	has the value (3, 4, <u>undefined</u>, <u>empty</u>, <u>empty</u>) (undefined, since that alternate in B1 has no value.)
SUBSCR (_, A1.2, A1.3)	has the value (<u>empty</u>, 20, NULL)
DESC (KIND = A1.1, AREA = 5, A. OFFSET1 = 10)	has the value (1, <u>empty</u>, (5, <u>empty</u>), (<u>empty</u>, 10, <u>empty</u>), <u>empty</u>)

6. Statements

```
<statement seq>        ::=   <statement> ;

<statement>            ::=   <open statement> | <closed statement>
<open statement>       ::=   <label> : <open statement>
                       ::=   <iterative open statement>
                       ::=   IF <expression> THEN <closed statement> ELSE <open
                             statement>
                       ::=   IF <expression>  THEN  <statement>

<closed statement>     ::=   <label> : <closed statement> | <empty>
                       ::=   <assignment statement> | <iterative closed statement>
                       ::=   <control statement> | <compound statement>
                       ::=   <case statement> | <code generation statement>
                       ::=   IF <expression> THEN <closed statement> ELSE <closed
                             statement> |<pass phase communication>
                       ::=   phase call
<label>                ::=   <identifier>
<compound statement>   ::=   BEGIN <statement seq> END
```

Statements are executed in sequence, as in ALGOL, except when a <control statement> is executed. The "IF" statements are as in ALGOL; the <expression> must yield a value which is TRUE or FALSE or else some arithmetic type (in which case <expression> $\neq$ 0 is implied). <compound statement>s are as they are in ALGOL.

<code generation statement>s are discussed in CGTM 105.

6.1 Assignment statements

Let a and b be simple variables and suppose we assign the value of b to a. The following gives the possible legal conversions and how they are executed. All values with type <bits> are assumed to be positive integers when used as arithmetic quantities.

a \ b	BYTE	BYTE2, 3,4.	HWI	FWI	FWF	DWF	DEC	POINTER	STRING
BYTE	BYTE	(a)	(b)	(b)	(d)	(d)	(d)		
BYTE2, 3, 4	(a)	(a)	(c)	(b)	(d)	(d)	(d)		
HWI	obvious	(f)	obvious	(e)	(d)	(d)	(d)		
FWI	obvious	obvious	obvious	obvious	(d)	(d)	(d)		
FWF	obvious	obvious	obvious	obvious	obvious	(g)	(g)		
DWF	obvious	obvious	obvious	obvious	obvious	obvious	?		
DEC	obvious	obvious	obvious	obvious	obvious	?	obvious		
POINTER								obvious	
STRING									obvious

(a) if b is larger than a, store only the <u>rightmost</u> bits of b into a.

(b) Put the rightmost i bits of b into a.

(c) Store b as it is, right adjusted with zeroes added on the left, into a.

(d) Truncate b to an integer, store in a.

(e) take the sign and rightmost 15 bits of b.

(f) Use the rightmost 16 bits of b, store in a (giving neg. or pos. number, depending on 16th bit).

(g) Use most significant part of b.

<assignment statement> ::= <data reference> .= <expression>

::= <data reference> .= <structure expression>

In the first case the <expression> is evaluated and the result stored into the <data reference> according to the beforementioned rules.

In the second case, those (sub)components of the evaluated <structure expression> which are not <u>empty</u> are stored into the corresponding (sub)-components of the <data reference>. Those (sub)components, whose corresponding components in the <structure expression> are <u>empty</u>, are not changed.

Suppose P1 is a variable of type pointer. Then

P1 ← @ <structure expression>

assigns to P1 the address of the place where the <structure expression> has been stored. It is up to the compiler writer to free the space allocated to the <structure expression> when he no longer needs it. See Section 6.7. Here, and below, after the assignment to P1 is made, <u>empty</u> (sub)components become undefined.

Suppose P is a pointer variable and suppose we have the assignments

(1) <P> ← <exp>

(2) P ← <exp>

(3) <P> ← @ <exp>

(4) P ← @ <exp>

The <exp> is evaluated — new storage space is allocated <u>only</u> if needed. For instance, if <exp> is just a <data reference>, no space need be allocated.

(1) takes the value of the <exp> and stores it at the location defined by the value of P. If space was allocated it is automatically released.

(2) assigns to P the value of the <exp> Thus, <exp> must have type POINTER. If space was allocated it is automatically released.

(3) puts the address of the location of <exp> into the location defined by the value of P.

(4) assigns to P the address of the location where <exp> is stored.

6.2 Iterative Statements

Let J be either the word "open" or "closed" for each production:

<iterative J statement> ::= FØR <data reference> .= $\boxed{\text{<for list element>}}$ DØ
<J statement>

::= WHILE <expression> DØ <J statement>

<for list element> ::= <expression>$_1$ $\boxed{\text{STEP <expression}_2\text{>}}$ UNTIL <expression>$_3$

The statement

FØR <data reference> .= <expression>$_1$ STEP <expression>$_2$ UNTIL <expression>$_3$ DØ S

is equivalent to

```
P1     .=   @ <data reference>
<P1>   .=   <expression>1 ;
T1     .=   <expression>2 ;
T2     .=   <expression>3 ⊗ SIGN (T1)

AGAIN:  IF <P1> ≤ T2
        THEN  BEGIN  S;  <P1>  .=  T1 + <P1>; GO TO AGAIN END
```

where P1 is a POINTER variable and T1 has the same type as <expression>$_2$ and T2 the same type as <expression>$_3$.

The statement

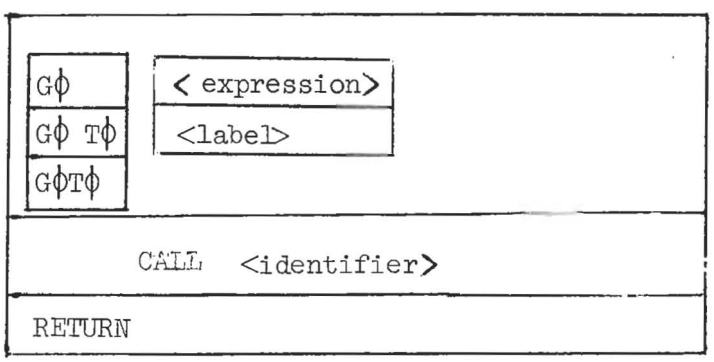
WHILE <expression> DØ S

is executed as

AGAIN: IF <expression> THEN BEGIN S; GO TO AGAIN END

6.3 Control statements

<control statement> ::=



Interphase and interpass communication is described in Section 6.5.

Gϕ, Gϕ Tϕ and GϕTϕ cause control to be given to the <label>. CALL <identifier> causes control to be passed to the procedure named. Control is returned to the point following this statement when the procedure is finished.

RETURN return from a procedure to the calling point of that procedure.

6.4 Case statements

<case statement> ::= CASE <expression> OF <statement seq> ENDCASE

The <expression> is evaluated and truncated to an integer, i. The ith <statement> in the <statement seq> is then executed. If this <statement> does not cause control to leave it completely, when the execution is finished control passes to the point beyond the symbol ENDCASE. If $i \leq 0$ or $i >$ (the number of statements), the case statement is skipped.

6.5 phase and pass communication statements

<pass-phase communication> ::= INIT <unsigned integer> | SCAN | HALT | SYNTAX

::= TURN (<unsigned integer>, $\boxed{\text{ON}}$ / $\boxed{\text{OFF}}$)

::= NEXTPASS (<unsigned integer>)

::= COMPLETE

::= BEGINAT ($\boxed{\text{SYN}}$ / $\boxed{\text{SEM}}$, <label>)

Each phase must have a procedure INIT without parameters. This procedure is used to do any necessary initialization for that phase. This pro-

cedure may execute any statements except SCAN, GO TO outside the procedure, a call of another procedure.

A pass begins with a call to the procedure INIT of the first phase. Upon return from this procedure, execution begins as directed through the use of the BEGINAT statement.

The execution of INIT <unsigned integer> causes the procedure INIT in the <unsigned integer>th phase to be executed.

SCAN is executed as follows.

Case 1. There is another phase, say the ith, in the sequence which is "on".

1. Put the symbol in SCANSYM onto the main stack of phase i (if it has one).
2. Give control to that phase following the last SCAN executed in it (or at the statement indicated by the last execution of procedure INIT of that phase).

Case 2. There are no more phases in the sequence which are "on".

1. Call the lexical analyzer to get the next source program symbol into SCANSYM.
2. Do as in case 1 with the first phase in the sequence which is on.

HALT stops translation and the job.

The TURN statement turns the <unsigned integer>th phase "ON" or "OFF". SYNTAX causes control to return to the SYNTAX subprogram at the point following: the last execution of EXEC (which got us into semantic subprogram in the first place).

The NEXTPASS statement calls in and begins the pass numbered <unsigned integer>.

COMPLETE signifies the end of translation and the begin of execution of the object program (if desired).

BEGINAT indicates that, when this phase is next executed, the execution should begin at the syntax or semantic statement labeled <label>.

6.6 Stack and table statements

<stack statement> ::= PUSH (<identifier> [, <exp>])

::= PØP (<identifier> [, <data reference>])

PUSH is used to add an element to a stack. This new element takes the value of the <exp> if there; otherwise it is undefined. PØP is used to delete the top stack element. If <data reference> is present, the element is stored into the data element referred to, before it is deleted.

Care must be taken when PUSHing and PØPing the main stack of a phase. A semantic routine should not PUSH and PØP if it refers later to L0, L1, ..., L6 and R0, R1, ..., R4.

<table statement> ::= TALLY (<table identifier>, <data reference> [, BACK[redacted] | ,FORWARD])

<data reference> must be of type pointer and its value must point to an element (say the ith) of <table identifier> or NULL. The pointer is changed to point to the $i+1^{st}$ element (FORWARD) or $i-1^{st}$ (BACK[redacted]). The default option is FORWARD. If FORWARD, and there are no more, the pointer is changed to NULL.

6.7 other statements

```
<other statement>  ::=   FAST <identifier>

                   ::=   NOFAST <identifier>
                   ::=   RELEASE <expression>
```

The metacompiler attempts no optimization on the object program produced. To allow the compiler writer some control, FAST and NOFAST are introduced. Between the statements FAST id and NOFAST id one (or two) register(s) will be used as the storage place of id. id must be name type <arith type>, <bits type> or <pointer type>.

The compiler writer must worry about jumps into space of the FAST statement; the following will not work.

```
LINE1:   P ← 2;  GΦ  LAB;

LINE2:   FAST  P;
               :
               :

     LAB:
          :
          :
          NOFAST  P;
```

since the value 2 will not be loaded into the register used for P. Interchanging LINE1 and LINE2 will solve the problem. At FAST P, P is loaded into the register. At a procedure call, the value will be stored in the location P and put back in the register upon return. At NOFAST P, the register is stored back into P.

With the RELEASE statement the locations defined by the value of the <expression> (of type POINTER) are released.

7. Alignment

7.1 Basic storage requirements

The following table gives the storage requirements and alignment for the simple types of variables.

type	space	leftmost byte address is a multiple of	number of bytes used
FWF	4 bytes	4	4
DWF	8 bytes	8	8
FWI	4 bytes	4	4
HWI	2 bytes	2	2
DEC	?	?	?
BYTE,BYTE2, 3, 4	1,2,3,4 bytes	1,2,4,4 (see a)	1,2,3,4
POINTER	3 bytes	4 (see a)	3 (see a)
STRING	4 bytes	4 (see b)	4

a) the value is contained in the last 3 bytes of the fullword addressed.

b) the first byte contains the number of characters in the string -0, 1, ... 255. the next three bytes points to the string itself.

7.2 STRUCTURE alignment

All components and subcomponents are stored using the rules above, and packed as close together as possible.

If the simple (sub)component needing the most space needs i bytes, then the structure itself starts on a byte whose address is a multiple of i. Thus, if the mentioned (sub)component is either HWI, BYTE or BYTE2 , then the structure address is a multiple of 2; if FWF or FWI, a multiple of 4; if DWF, a multiple of 8.

The same holds for components and (sub)components which are structure types, of course.

Space is allocated to the components in a left-right manner, following the rules mentioned above and in 7.1, first for the 1st component, then the 2nd, etc. All alternates of a component use the same space.