SLAC-102
UC-32
(MISC)

CIL

COMPILER IMPLEMENTATION LANGUAGE

DAVID GRIES

STANFORD LINEAR ACCELERATOR CENTER

STANFORD UNIVERSITY

Stanford, California

PREPARED FOR THE U.S. ATOMIC ENERGY

COMMISSION UNDER CONTRACT NO. AT(04-3)-515

March 1969

CONTENTS Revised 3/10/69

1. INTRODUCTION TO CIL

This report is a manual for the proposed Compiler Implementation Language, CIL. It is not an expository paper on the subject of compiler writing or compiler-compilers. The language definition may change as work progresses on the project.

1.1. Basic features of CIL

The Compiler Implementation Language is designed for writing compilers for the IBM 360 computers. The heart of the system is a procedure oriented ALGOL-like language with expressions, assignment statements, iterative statements, etc. However the basic data types of the language are those of the IBM 360 - byte, halfword integer, sequence of 1 to 256 bytes, etc - while the basic operations on these types of data are also those of the 360. This should allow the compiler writer to have more feeling for the code generated by the metacompiler and thus make it possible to write more efficient compilers.

In addition, the following features are provided to facilitate compiler writing:

1. Scanner definitions. A compiler writer declares the source language symbols (reserved words, operators, format of identifiers, etc.) in a scanner definition. From this the metacompiler builds an efficient scanner which, at compile-time, will read a source program, break it up into these symbols and pass them one at a time to the compiler itself.

   The scanner definition has been designed to handle most of the existing languages. It has however been restricted so that efficient scanners can be built. Should it be necessary, the compiler writer can inspect the string of characters making up any symbol and/or switch to a character-by-character scan, in which case he may form his own symbols.

2. Atoms. A hash-coded internal dictionary of all source language symbols is kept current as a source program is read by the scanner. This dictionary is used to replace each symbol by a 16 bit representation called an atom. It is this atom that is passed to the compiler by the scanner. The compiler automatically uses these fixed length atoms instead of the variable-length source language symbols. In this report, "source language symbol" and "atom" are used synonymously.

3. Production language (PL). This is a sublanguage for performing the syntax analysis of source programs. It consists of "Floyd productions", each of which attempts to match certain symbols with the top symbols of a last-in-first-out (LIFO) stack. When a match occurs, "actions" in the production change the stack and cause "semantic routines" to be called in order to process the symbols matched.

4. Structured types. A programmer can define his own structured

types; these are sequences of components, analogous to the WIRTH and HOARE records. In order to save space, several alternates can be declared for each component. Once defined, variables of a structured type can be declared in the same way as usual variables are declared.

5. Tables, dicts and stacks. These are all sequences of records; the difference is in the way the records are accessed. No upper bound on the number of records need be given. The records themselves may have a structured type (see (4) above).

   Records of a dict are chained to records of the internal dictionary (see (2) above) to provide fast searches of records based on source language identifiers.

6. Multiple coreloads. A compiler can consist of any number of coreloads, which are executed in a fixed order. Thus, both single-pass compilers and compilers which perform sophisticated transformations and code optimization can be written.

7. Code generation. This is the most important addition to the language. Our code generation system (CGS) is based on Feldman's "code bracket" scheme [Comm. Of the ACM, Vol. 9, Jan. 1966] . The purpose is to give the compiler writer a high-level language for generating IBM 360 machine language. The compiler writer should be familiar with the IBM 360 data types and the instruction set. However he can leave register allocation, storage allocation, generation of instructions, conversion of runtime operands, etc. To the system.

   The basic features of this system are:

   A. CODEAREAS and DATAAREAS. A compiler writer may generate code into any number of CODEAREAS (read-only storage at runtime) and may use any number of DATAAREAS (read-write storage). This ability to use different CODEAREAS (one for each subroutine, say) and DATAAREAS (one for the variables associated with each subroutine,say) simplifies the compiler writer's task. Most problems connected with addressing code or data in these AREAs are handled by CGS.

   B. Register descriptions. CGS maintains register descriptions describing the runtime state of the IBM 360 registers after the last-generated instruction has been executed. CGS performs some local code optimization with the help of the register descriptions. The descriptions may also be tested and changed by compiler writer.

   C. DESCRIPTORS. DESCRIPTORS are used to describe runtime variables in terms of the basic data types of the IBM 360, such as byte, halfword integer and fullword integer. The runtime address of a variable is described by a CODE or DATAAREA number and an offset into the AREA. The DESCRIPTOR can also indicate up to two levels of indirect addressing and/or subscripting. The DESCRIPTOR also

contains information such as whether the value is in a register, whether it is a constant, etc.

D. Storage allocation and initialization. Primitives exist for allocating storage in CODE or DATAAREAS for runtime variables. Problems or correct alignment and the like are handled by CGS. In certain cases the allocated storage can be initialized.

E. Code brackets. In general, any statement or expression may appear between the code brackets "CODE (" and ")". This indicates that the statement or expression is to be executed at runtime. The operands of the statement or expression must be DESCRIPTORS (of runtime variables), constants, or variables declared to be valid at runtime. For example, suppose D1 and D2 are DESCRIPTORS of an integer variable and an array element, respectively. Then execution of

```
CODE( FOR D1 = 1 UNTIL 10 DO D2(D1) = 5)
```

would generate code to set the first 10 elements of the array to 5.

When a code-bracket statement is executed, code is generated into the current CODEAREA as specified in the statements or expressions within the code brackets, and the register descriptions for that CODEAREA are changed to describe the new runtime state of the registers. CGS also automatically generates code for any necessary conversions between data types.

All the additional features of CIL need not be used. For example,

1. An interpreter could be written without the use of the code generation system; a first pass could put the program in an intermediate form and a second pass could then interpret it.

2. Production language need not be used; any type of syntax analyzer can be programmed using the normal ALGOL-like constructs of the language.

3. The language can be used for writing "normal" programs. Throw out the scanner definition, PL, and CGS and an ALGOL-like language remains. The basic data types of the language and the operations on them are those of the IBM 360 computer; this high-level language just provides a convenient tool for using them.

1.2. How to read this report

The best way to get acquainted with the language is to read the

program examples in Appendix C. You will find that CIL is basically an ALGOL - like procedural language. Then read Sections 2 through 10 which describe this procedural language and its normal use. Skip over references to the scanner definition, PL or CGS. Finally, read the three additional sections 12 (on the scanner definition), 13 (on PL) and 14 (on CGS).

1.3. Acknowledgements

Sheldon Becker, Lee Erman, Gary Goodman, Lockwood Morris, Jim Cook and Christiana Riedl have all programmed or are programming parts of the system. All of them have contributed to the language and this manual. Thanks also go to Jerry Feldman for his useful thoughts on the subject.

2. TERMINOLOGY AND NOTATION

2.1. Definitions

Metacompile_time is the time during which a compiler - or any program written in CIL - is being compiled.

Compile_time is when a source program is being compiled by a compiler written in CIL.

Runtime is when a compiled source program is being executed.

A source_program is a program written in a source language.

Source_language refers to the language for which a compiler has been written in CIL.

2.2. Syntax notation

Backus Normal Form (BNF) with some modifications will be used to describe the syntax of this programming language. Syntactic class names (nonterminal symbols) are enclosed in angular brackets "<" and ">", while the symbols of the language (terminal symbols) are represented by themselves. A production consists of a left_part, which is always a syntactic class name, followed by the metasymbol "::=", followed by a right_part - one or more syntactic class names or terminal symbols. It indicates that the syntactic class given by the left part consists of those strings of symbols described by the right part. Thus the productions

```
<identifier> ::= <letter>
<identifier> ::= <identifier> <letter>
<identifier> ::= <identifier> <digit>
```

indicate that an identifier consists of a letter or another identifier followed by a letter or digit. In other words, an identifier is a letter followed by zero or more letters or digits. As an abbreviation, the metasymbol "|" is used to write the above three productions as

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

Thus "|" is used to separate right parts of productions whose left parts are the same.

The following modifications to BNF have been introduced to provide a clearer syntactic description.

1. The right part of a production may be partly described by a comment enclosed in quotes. Thus we write

```
<string>     ::= ' "sequence of 1 to 256 EBCDIC characters" '
```

2. In order to prevent misinterpretation, the source symbols "<" and ">" will always be enclosed in quotes. Thus we write

<relation> ::= <expression> "<" <expression>

3. Square brackets are used to enclose optional entities. For example,

<factor> ::= [<unary op>] <primary>

is equivalent to

<factor> ::= <primary> | <unary op> <primary>

4. The nonterminal symbol <empty> represents the empty string.

5. A sequence of one or more symbols, all belonging to the syntactic class <x>, can be written as <<x> list>. If they are to be separated by a terminal symbol, then this terminal symbol directly precedes the word "list". Thus

<basic decl> ::= <basic type> <<identifier> ,list>

is exactly equivalent to

<basic decl> ::= <basic type> <id list>
<id list> ::= <identifier> | <id list> , <identifier>

and

<integer> ::= <<digit> list>

is equivalent to

<integer> ::= <digit> | <integer> <digit>

6. If a nonterminal appears more than once in a production, the occurrences may be numbered so that they can be identified in the semantic discussion. Thus we write

<for list> ::= <expression¹> UNTIL <expression²>

7. The syntactic classes <specfunc> and <specproc> denote special function designators and special procedure calls respectively. The syntax of these <specfunc>s and <specproc>s is always given in boxes. For example,

```
| PUSH ( <stack identifier> [ ,<exp> ]) |
```

2.3. Syntactic entities (with corresponding numbers)

3. THE BASIC ELEMENTS OF THE LANGUAGE

3.1. Basic symbols, comments and spaces

Syntax

```
<basic symbol>    ::= <letter> | <digit> | <delimiter>

<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 | &

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

<delimiter>       ::= + | - | * | / | ¬ | = | @
                    | . | , | ; | ' | ( | ) | :
                    | "<" | ">" | "<=" | ">="
                    | "¬<" | "¬>" | ¬=
                    | _ | $ | /* | */ | // | **

<EBCDIC char>     ::= "any EBCDIC character except space"
```

Semantics: Letters are use for forming identifiers and reserved words. Digits are used in forming numbers and identifiers. Bits and hexits are used in forming constants. The meaning of delimiters will be given at the appropriate place in the sequel.

Except in a PL subprogram and a scanner definition, a comment of the form

/* "any sequence of characters not including "*/" " */

may appear anywhere. It is the equivalent of a single space.

Changing to a new card or line has no significance. Outside of strings, spaces have no meaning except for the following rules:

1. At least one space must separate two adjacent identifiers, <source language symbol>s (cf Section 3.4), integers or reserved words.

2. A space may not separate two characters of a delimiter, identifier, integer, reserved word or source language symbol.

This section has defined the characters used in writing a compiler in CIL. This does not preclude the use of other characters or the use of these characters in a different way in a source language for which a compiler is being written.

3.2. Identifiers and integers

Syntax

```
<identifier>        ::= <letter> | <identifier> <letter>
                      | <identifier> <digit>
<integer>           ::= <<digit> list>

<component id>      ::=<identifier>
<dict identifier>   ::=<identifier>
<int identifier>    ::=<identifier>
<label>             ::=<identifier>
<scanner id>        ::=<identifier>
<stack identifier>  ::=<identifier>
<structured type>   ::=<identifier>
<synonym>           ::=<identifier>
<table identifier>  ::=<identifier>

<pass number>       ::= <integer> "between 1 and 25"
```

Semantics: Integers have their conventional meaning as decimal numbers. Identifiers have no inherent meaning but serve to identify variables, labels, procedures, structure types, and scanner definitions. They may be chosen freely except that they may not also be reserved words of the language (cf Section 3.3). In addition, several identifiers are already implicitly declared by the system. They may be declared in a program, but this precludes their use as system identifiers (cf Appendix B). Note that the letter & may be used in an identifier. Many system identifiers begin with & and it would be wise to refrain from using & in this way.

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf Sections 6 and 4.2).

The recognition of the definition of a given identifier (but not a component identifier -cf Section 7) is determined by the following rules.

Step 1. If the identifier is defined by a declaration of a quantity or structure type, or is standing as a label within a procedure embracing the occurrence of the identifier, then it denotes that quantity, structure type, or label.

Step 2. Otherwise, if the identifier is a formal parameter of a procedure embracing the occurrence of the identifier, then it stands for that formal parameter.

Step 3. Otherwise, if the identifier is defined by a declaration of a quantity or structure type or by its standing as a label within a pass embracing the occurrence of the identifier, then it denotes that quantity, structure type, or label.

Step 4. Otherwise, if the identifier is defined by a declaration of a quantity or structure type in a global declaration valid in the pass (or global declaration) embracing the occurrence of the identifier, then it stands for that

quantity or structure type.

Step 5. Otherwise, if the identifier was declared as a <synonym> in a scanner definition, then it stands for the corresponding source language symbol.

If any single step could lead to more than one definition, then the identification is undefined.

3.3. Reserved words

The following reserved words may not be used as identifiers.

```
ALT AND
BACK BEGIN BITAND BITEXOR BITOR
BYTE BYTES BYTE2 BYTE3 BYTE4
CASE CODE CODEAREA CONTENT CORELOAD
DATAAREA DEC DELETE DICT DO DWF DYNAMIC
ELSE END ENDCASE ENDPASS ENTER
FOR FROM FWF FWI
GO GOIF GOIFNOT GOTO
HWI
IF IN
LOOK
MAIN
NOT
OF OR
PASS PASSES POINTER POP PROCEDURE PRODLANG PUSH
REM RETURN RUNTIME
SCANNER STACK STATIC STRING STRUCTURE
SUBBYTE SUBSTR SYNTAX
TABLE TALLY THEN TO
UNTIL
WHILE
&C
```

3.4. Source language symbols

Syntax:

```
<source language symbol> ::= <synonym>
                           | $ <<EBCDIC char> list>
```

Semantics: A source language symbol is a sequence of characters defined in a scanner definition to be a delimiter or reserved word of the language for which a compiler is being written. One refers to the BYTE2 atom for a source language symbol either by preceding it by a dollar sign, or by using a synonym for it (cf Section 12.2). No space may separate the dollar sign from the character list or the characters in the list themselves and a space must follow the last character.

4. STRUCTURE OF A PROGRAM

Syntax:

```
<program>          ::= BEGIN [ <<declaration> ;list> ]
                       [ <<statement> ;list> ] END

<program>          ::= BEGIN <coreload description>
                       [ <<scanner def> list> ]
                       [ <<global declaration> list> ]
                       <<pass> list>
                       END
```

Semantics: The first definition of a program is for the usual ALGOL-like program consisting of declarations (cf Section 6) and statements (cf Section 9). The second must be used for programs with multiple passes or programs which use a scanner or production language.

4.1. Coreload description

Syntax:

```
<coreload description> ::= <<coreload> list>
<coreload>             ::= CORELOAD <integer>
                           <<pass number> list>
```

Semantics: The coreload description indicates how storage is to be allocated to the passes of a compiler. The coreloads must be numbered (by the <integer>) in ascending order, starting with 1. At compile time, initially all the passes associated with coreload 1 are in core, and the first pass listed is executed. Upon execution of a CALLPASS statement (cf Section 9.6) which refers to a pass in a different coreload, the new coreload is brought into core. The passes in the previous coreload may not be referred to again.

4.2. Global declarations

Syntax:

```
<global declaration>   ::= PASSES <integer¹> <integer²>
                           <<declaration> ;list>

                           | PASSES <integer¹> RUNTIME
                           <<declaration> ;list>

                           | RUNTIME <<declaration> ;list>
```

Semantics: A global declaration declares identifiers (and their attributes) which are to be used globally in

a) passes numbered <integer¹> through <integer²>;
b) passes <integer¹>, <integer¹> + 1,..., and at runtime;
c) at runtime only.

The following restrictions are placed on identifiers declared in a

global declaration

a) no identifier may be a reserved word (cf Section 3.3);
b) the same identifier may not be declared in two global declarations which have a pass in common. Thus

```
PASSES 1 4 BYTE A,B
PASSES 2 3 BYTE B,C
```

is illegal;
c) an identifier must be declared before it can be used.

Declarations themselves are discussed in Section 6.

Examples:

```
PASSES 1 2 BYTE A,B,C; POINTER P
PASSES 5 RUNTIME STRING X
RUNTIME BYTE Y; FWI A,B
```

4.3. Passes

Syntax:

```
<pass>          ::= PASS <pass number> [<PL subprogram>]
                    [ <<declaration> ;list> ]
                    [ <<statement> ;list> ]
                    ENDPASS
```

Semantics: A pass is a logical unit - a subprogram. Section 9.6 discusses the statements which control the order of execution of passes. When a pass begins, if no PL subprogram is present, the first statement in the list is executed. If a PL subprogram is present, execution begins with the first production in it.

5. VALUES, TYPES AND CONSTANTS

A variable is a symbolic representation of a quantity that may assume different values . The value of a variable is always the one most recently assigned to it. Each variable has a type which defines the class of values that the variable may represent.

Types fall into two classes: basic types - which are the basic, elementary types in the language - and structured types - which are ordered sets of one or more basic types and possibly other structured types. Structured types are defined by the programmer in a structure definition.

The number of bytes each different type of value uses in the IBM 360 and the alignment of these bytes in memory are discussed in Section 11. Section 5.1 describes the basic types in the language, Section 5.2 structured types and the structure definition. Constants are described in Section 5.3.

Syntax:

```
<type>          ::= <basic type> | <structured type>
```

5.1. Basic types

Syntax:

```
<basic type>    ::= <bits type> | <arith type>
                  | <pointer type> | <string type>

<bits type>     ::= BYTE | BYTE2 | BYTE3 | BYTE4
                  | BYTES ( <integer> )
<arith type>    ::= HWI | FWI | FWF | DWF | DEC
<pointer type>  ::= POINTER
                  | POINTER ( <<pointo type> list> )
<string type>   ::= STRING ( <integer> )

<pointo type>   ::= <bits type> | <arith type> | POINTER
                  | <string type> | <structured type>
```

Semantics: The types BYTE, BYTE2, BYTE3 and BYTE4 are essentially abbreviations for BYTES(1), BYTES(2), BYTES(3) and BYTES(4), respectively. Note however the different alignment properties (cf Section 11).

The following table lists the values that may be associated with a variable of each basic type.

type	Value
BYTES(<integer>)	sequence of 8*<integer> bits (0 < <integer> <= 256)

HWI	IBM 360 HalfWord Integer: 16 bits (between -2**15 and 2**15-1)
FWI	IBM 360 FullWord Integer: 32 bits (between -2**31 and 2**31-1)
FWF	IBM 360 FullWord Floating point number: 32 bits
DWF	IBM 360 DoubleWord Floating point number: 64 bits
DEC	DECimal number of 1 to 31 digits plus sign
STRING(<integer>)	sequence of <integer> EBCDIC characters (0 < <integer> < 256)
POINTER	reference to some value(24 bit address)

When referring to the value pointed at by a variable declared as POINTER, it is necessary to indicate what type that value has. This can be done at the point of referral (cf Section 7.3), or in the declaration itself through the list of <pointo type>s. For example,

POINTER A	A may point at any value.
POINTER(FWF) B	B may only point at values of type FWF.
POINTER(FWF HWI)C	C may point at values of type FWF and HWI.

Hierarchy of types. It is sometimes necessary to perform automatic conversion of values. For example, if one adds an FWI value to an FWF value, the FWI value must first be converted to floating point form. The hierarchy of type precedences is:

DWF
FWF
DEC
FWI
HWI
BYTES

5.2. Structured values and types

Syntax:

```
<structure definition>
                    ::= STRUCTURE <structured type>
                        ( <<constituent> ,list> )

<constituent>       ::= <component>
                      | <constituent> ALT <component>

<component>         ::= <type> <component id>
                      | <component id> (<<constituent> ,list>)
```

Semantics: A structure definition defines a new structure named <structured type>. A structured value is a set of constituents - which at any instant of runtime are values with basic types and possibly other structured types. Each constituent consists of a single component or it consists of a set of alternative components separated by the reserved word ALT . This is used mainly to save space. Only one of the alternative components may be in use at any time, and it is the responsibility of the programmer to know which one is being used.

The name of each component is the component id. This name is used to refer to that component of the structured type. The component id may be any valid identifier which is not a structured type; the only rule to be followed is that, when referring to components and subcomponents of a structured value, the metacompiler must be able to uniquely determine what is meant. See Section 7.2 for full details.

Note that a component may itself contain subcomponents. If a structured type is used as the type of some component, this structured type must have been previously (statically) declared.

While not necessary, it may be useful for the programmer to know how storage is allocated to components. This is discussed in Section 11.

Examples:

1. STRUCTURE SUBSCR (BYTE AREA, BYTE3 OFFSET, POINTER S)

A value of type SUBSCR consists of

a) a BYTE value named AREA , followed by
b) a BYTE3 value named OFFSET , followed by
c) a POINTER value named S.

2. STRUCTURE D1 (BYTE KIND ALT HWI B, C (BYTE C1, POINTER C2),
SUBSCR D, SUBSCR E)

A value of type D1 consists of

a) EITHER a BYTE value named KIND
or a halfword integer named B, followed by
b) a value named C. C itself consists of
1) a BYTE value named C1 followed by
2) a POINTER value named C2.
C is followed by
d) a value, named D, of structured type SUBSCR
e) a value, named E, of structured type SUBSCR

5.3. Constants

Syntax:

```
<constant>          ::= <integer> | <hex integer>
                      | <bit integer> | <dec integer>
                      | <real> | <long real>
                      | <logical cons>
                      | <string cons> | <pointer cons>
                      | <synonym> | <int identifier>

<hex integer>       ::= X ' <<hexit> list> '
<bit integer>       ::= B ' <<bit> list> '
<dec integer>       ::= <integer> D
<real>              ::= <unscaled real> [ <scale factor> ]
<long real>         ::= <real> L
<string cons>       ::= ' "sequence of 1 - 256 EBCDIC
                      characters" '
<pointer cons>      ::= 0

<unscaled real>     ::= <integer> . <integer> | <integer> .
                      | . <integer>
<scale factor>      ::= E <sign> <integer>
<sign>              ::= + | -
```

Semantics: Integers, reals and long reals are interpreted according to the conventional decimal notation. A scale factor denotes an integral power of 10 which is multiplied by the unscaled real preceding it. A dec integer is an integer of 1 to 31 digits which will be represented in packed decimal notation.

A string constant is a sequence of 1 to 256 characters, enclosed by the string quote " ' ". Within the sequence, the string quote itself is to be represented by two adjacent string quotes. The number of characters in the string is called the length of the string.

Each hexit in a hex integer represents 4 bits in the usual manner. Both hex integers and bit integers are right adjusted in their field, with leading zero bits added if necessary (see below).

The pointer cons 0 fails to point to a value.

A synonym denotes the atom corresponding to the source language symbol associated to the synonym in a <synonym def> of the scanner sublanguage (cf Section 12.2).

An int identifier is a BYTE2 constant. The actual value is assigned by the metacompiler (see Section 6.4).

Each constant has a unique type, as defined by the following list. It should be noted that any necessary conversion of constants is done at metacompile time when possible.

<constant>	<type>
<integer>	HWI if less than 65536. FWI otherwise
<hex integer>	BYTES(I), where if there are J hexits, 2*I >= J > 2*I-2
<bit integer>	BYTES(I), where if there are J bits, 8*I >= J > 8*I-8
<dec integer>	DEC
<real>	FWI
<long real>	DWI
<string cons>	STRING (<integer>)
<pointer cons>	POINTER
<synonym>	BYTE2
<int identifier>	BYTE2

In addition, the following system identifiers for constants can be used.

TRUE	BYTE1 (=X'FF')
FALSE	BYTE1 (=X'00')

Examples:

<constant>	examples			
<integer>	1	23	325678	
<hex integer>	X'0A'	X'B32A'	X'FFFFFFFF'	
<bit integer>	B'0110'	B'100100100000'		
<dec integer>	32D	100D	1357312389D	
<real>	3.	.56	32.031	3.E-20
<long real>	2.718281828459045235	3L	.3E-1L	
<string cons>	'STRING'	'0'	'''' is the string consisting of a single apostrophe.	

6. DECLARATIONS

Declarations serve to determine the scope of identifiers and to define permanent properties of them (type of value that may be associated with them, structure). Generally, a number of bytes are allocated to each identifier (depending on the type) to hold the value associated with it. See Section 11 for full details.

Syntax:

```
<declaration>        ::= <structure definition>
                       | <type dec>
                       | <int declaration>
                       | <table declaration>
                       | <dict declaration>
                       | <stack declaration>
                       | <main stack dec>
                       | <procedure declaration>
```

6.1. Basic and structured type declarations

Syntax:

```
<type dec>           ::= <basic type dec>
                       | <structured type dec>
<basic type dec>     ::= <basic type> <<identifier> ,list>
<structured type dec> ::=<structured type> <<identifier> ,list>
```

Semantics: Basic and structured type declarations serve to associate a type with identifiers. Only values of that type may be assigned to the identifiers. The structured type must have been previously (statically) declared.

Examples:

```
FWI A,B,C
POINTER (SUBSCR) D   (see Section 5.2 for the structure
                      definition for SUBSCR).
SUBSCR E,F,G
```

6.2. Table, dict and stack declarations

Syntax:

```
<table declaration> ::= <storage alloc> <type> TABLE <integer>
                        <table identifier>
                      | STRING TABLE <table identifier>
                        = <<string cons> ,list>

<dict declaration>  ::= <storage alloc> <type> DICT <integer>
                        <dict identifier>
<stack declaration> ::= <storage alloc> <type> STACK <integer>
                        <stack identifier>
```

<main stack dec> ::= MAIN STACK <stack identifier>

<storage alloc> ::= STATIC | DYNAMIC | <empty>

Semantics: Table, dict and stack declarations all serve to associate a sequence of data records of type <type> with the table, dict or stack identifier. The difference is only in the way the records are added, deleted or accessed. See Section 10.0 for full details:

A table is a linear sequence of records. Records are usually accessed through pointers to them and by the operations LOOK and ENTER. They may however be accessed exactly like a one dimensional ALGOL array.

A dict is also a sequence of records, these records are however list-structured for fast searches based on source language symbols. Records may be added to or deleted from the dict. They may also be taken off the chain which list-structures them. The type of the dict records must be a structured type. Further, the structured type, say T, must begin as follows:

STRUCTURE T (BYTE NAME1, POINTER NAME2, ...

Here, the component ids are not important; only the fact that the first two components are a BYTE and a POINTER. The reason for this will become clear when Section 10.2 on LOOK and ENTER is read.

A stack is a LIFO (last-in-first-out) stack. Records may be added and deleted in the customary manner.

<storage alloc> indicates how storage is to be allocated to the sequence. If STATIC or <empty>, <integer> gives the maximum number of records in the table, dict or stack. These records will be contiguous. If DYNAMIC, <integer> defines the number of contiguous records in a "block". Storage is initially allocated to one block of records; extra blocks are added as the need arises while the program is being executed.

Each pass which uses production language must have a stack to communicate between the production language and semantic language. This stack is specified by a <main stack dec>. The stack identifier in the <main stack dec> must be a previously declared STATIC stack. In addition, the type of the stack records must be a structured type, say S, which begins as follows:

STRUCTURE S (BYTE2 NAME1, BYTE2 NAME2, BYTE2 NAME3, ...

Here, the component ids are not important; only the fact that the first three components are BYTE2 quantities. See Section 13.6.

Examples:

```
SUBSCR TABLE 200 A
DYNAMIC D1 DICT 50 B
STAIIC D1 STACK 100 C
MAIN STACK C
```

6.3 procedure declarations

Syntax:

```
<procedure declaration> ::= PROCEDURE
                            <procedure heading> ; <procedure  body>
                          | <type> PROCEDURE
                            <procedure  heading> ; <procedure body>

<procedure heading> ::= <identifier>
                          [ ( <<formal parameter seg> ;list>) ]
<formal parameter seg>
                    ::= <type specifier> <<identifier> ,list>
<type specifier>    ::= <type> | BYTES | STRING
                          | <type> TABLE | <type> DICT
                          | <type> STACK

<procedure body>    ::= <statement>
                          | BEGIN [ <<type dec> ;list> ]
                            [ <<statement> ;list> ] END
```

Semantics: A procedure declaration associates a procedure body with the identifier immediately following the symbol PROCEDURE. A proper procedure (case 1 above) is invoked by a procedure statement (cf Section 9.7) and a function (typed procedure - case 2 above) by a function designator (cf Section 8.1) or a procedure statement.

The procedure heading also describes the formal parameters and their types. All formal parameter identifiers in a formal parameter segment are of the same indicated type. The type specifiers BYTES and STRING specify formal parameters whose corresponding actual parameters at a call point are BYTES(I) and STRING(I) for some integer I. It is more efficient to indicate the number of bytes if it is constant for all calls of the procedure or function.

The value to be returned by a function is indicated by assigning it to the function identifier.

Examples:

```
PROCEDURE LOOKLAB(BYTE2 ATOM; POINTER P);
/* LOOK IN SYMBOL TABLE SSYMB FOR THE SOURCE SYMBOL "ATOM" WHCH
   IS A LABEL. RETURN THE ADDRESS OF THE RECORD IN P.*/
     BEGIN P = LOOK(SSYMB,ATOM);
           WHILE P ¬= 0 DO BEGIN IF P.TYPE = LABEL
```

```
                    THEN RETURN;
                    P = LOOK(SSYMB,P)
                END
        END
```

6.4. Int declarations

Syntax
<int declaration> ::= INT <<identifier> ,list>

Semantics: In production language an INT is a nonterminal or INTernal symbol used to help parse the program. In order to allow the semantic portion of a compiler to test the main stack and to provide more communication between syntax and semantics, the int declaration has been provided. Each identifier declared as INT is a BYTE2 constant - the actual value being assigned by the metacompiler. It may be used anywhere a constant may be used (cf Section 13).

7. VARIABLES AND INDIRECT REFERENCES

In Section 5 we described the different types of values possible. In Section 6 we indicated how these types could be associated with identifiers. We now describe how one references the value associated with an identifier - either to use it or to change it.

```
Syntax:
<destination>        ::= <variable> | <indirect reference>
<DESCR destination>  ::= <destination> "of type DESCRIPTOR"

<variable>           ::= <simple variable>
                       | <component variable>
<simple variable>    ::= <identifier>
                       | <table designator>
                       | <dict designator>
                       | <stack designator>
                       | <substring designator>
                       | <subbyte designator>

<component variable>::= <simple variable>.<component selector>


<indirect reference>::= CONTENT( <POINTER expr>
                         [ <pointo type> ] )
                       | <variable> . <component selector>

<table designator>   ::= <table identifier> ( <expression> )
<dict designator>    ::= <dict identifier> ( <expression> )
<stack designator>   ::= <stack identifier> ( <expression> )
                       | L0 | L1 | L2 | L3 | L4 | R0 | R1 | R2
<substring designator> ::= SUBSTR ( <destination>
                         , <expression> [ , <expression> ] )
<subbyte designator>::= SUBBYTE ( <destination>
                         , <expression> [ , <expression> ] )

<component selector>::= <<component id> .list>
                       | <<number selector> .list>
<number selector>    ::= <integer> [ ¬ <integer> ]
```

7.1. Simple variables

A table designator denotes a record of a table. The expression is evaluated, assigned to an internal integer variable I (say), and the Ith record is chosen. The value I must be greater than 0 and, if the table is STATIC, less than or equal to the number of records declared.

The time necessary to calculate the address of a record T(I) is directly proportional to the number of the block in which the record resides.

The usual way of accessing table records is through the LOOK

and ENTER commands and through POINTER variables which point at the records. If these commands are used, the following restriction is placed on the used of table designators: the value of I must always select an already-existing record; if not, an error may result. This is not checked at runtime.

If ENTER, LOOK and DELETE are not used, then the table is actually a one dimensional array. If it is declared DYNAMIC, then it may have any number of records. Thus, if a value I is used but there are not as yet I records in the table, enough blocks of records are added to yield I of them.

A dict designator denotes a record of a dict. This works exactly like a table designator.

A stack designator references a stack record. The expression is evaluated, assigned to an internal integer variable I, and the Ith record from the top of the stack is chosen. Thus, if S is a stack, S(0) refers to the top record, S(1) the first from the top, etc. If a pass has a main stack, then the system identifiers L0,...,L4 refer to the top main stack record,..., 4th record from the top of the main stack, before matching of the last production began, while R0,R1, AND R2 refer to the current top, 1st and 2nd records of the main stack, respectively.

A substring designator denotes a sequence of characters of the string <destination> the first expression is evaluated and assigned to an internal integer variable I. I then selects the position in the <variable> of the starting character of the sequence. The first character has position 0. Thus we have 0 <= I < declared length of the string variable. The second expression is evaluated and assigned to an internal integer variable J. J is then used as the length of the selected sequence. I+J must be less than or equal to the declared length of the string variable. The default value for the second expression is (length of string variable -I).

A subbyte designator denotes a sequence of bytes of a BYTES variable or indirect reference. The semantics are the same as those of substring designators.

7.2. Component variables and selectors

A component variable references a component of some structured variable. The first syntactic entity in a component variable is a simple variable, which chooses the particular structure from which the component is to be taken. This is followed by a period and a component selector, which picks out the desired component. There are two methods for this - naming the component, or indicating its position by a sequence of numbers.

A. Naming the component. The component selector is a sequence of component identifiers, separated by periods. The first is the name of a component of the structure. If there is only one

component identifier, then the desired component has been found. If there are more, then the first must name a component which itself has subcomponents. The second name picks out the desired subcomponent, etc. As an example, consider the declarations

```
STRUCTURE SUBSCR (BYTE AREA, BYTE3 OFFSET, POINTER S);
STRUCTURE D1 (BYTE KIND ALT HWI B,
                C (BYTE C1, POINTER C2),
                SUBSCR D, SUBSCR E);
D1 A;
```

To pick out component B of A use A.B .
To pick out component C1, use A.C.C1 .
To pick out component S or component D of A, use A.D.S .

It is not always necessary to give the complete list of component ids. Thus, in the above examples, A.C1 is equivalent to A.C.C1. The only rule is that the component variable must unambiguously define a component. A.S Would not be valid, since it could be either A.D.S Or A.E.S.

B. Numbering the component. Constituents are numbered from the left, starting with 1. Within a constituent, the alternate components are similarly numbered. A number selector I selects the first component of the Ith constituent. Thus we have:

A.1 equivalent to A.KIND
A.2 equivalent to A.C
A.2.1 equivalent to A.C.C1 .

How would we reference component B? By A.1¬2. Here, the "¬2" specifies the particular alternate (the second). In general, "I¬J" means, the Jth alternate for the Ith (sub)constituent. As illustrated above, A.1 is equivalent to A1.¬1.

7.3. Indirect references

A simple reference

CONTENT (<POINTER expression>)

references the variable "pointed at" by the POINTER expression. Thus, using the examples of the preceding section, if PP is a pointer variable, then executing

PP = @ A.KIND; CONTENT(PP) = 3

sets the component A.KIND to 3 (cf Section 8.2.1). The reserved word "&C" can be used as an abbreviation for "CONTENT".

It is necessary to indicate what type of value is being pointed

at, by including a <pointo type>. This may of course be done in the declaration of a POINTER variable (cf Section 5.1), in which case it can be left out here. The above example could be written as

```
PP = @ A.KIND; CONTENT(PP BYTE) = 3
```

If a POINTER expression points at some structured type value, then one can designate a component or subcomponent of that value exactly as was explained in Section 7.3.

Again, the <pointo type> may be omitted here if it is possible to determine from the component selector which structured type is being referred to. Thus, using the examples of Section 7.2, if there is no other structure with a component named C, CONTENT(PP).C could be used instead of CONTENT(PP D1).C.

As a further simplification - one which should be used often - if the POINTER expression is just a variable, and if the <pointo type> can be omitted, then the contents brackets can also be omitted. We could thus write PP.C for CONTENT(PP D1).C and PP.C.C2 for CONTENT(PP D1).C.C2.

7.4. Examples

syntactic entity	example
<identifier>	A
<table designator>	T(I+J)
<dict designator>	D(N)
<stack designator>	S(0)
<substring designator>	SUBSTR(ST,5)
<subbyte designator>	SUBBYTE(SY,5,I)
<component variable>	D(N).C.C2
	A.S
<indirect reference>	CONTENT(P SUBSCR)
	CONTENT(P)
	CONTENT(P SUBSCR).AREA
	CONTENT(P).AREA
	&C (&C (P SUBSCR).S BYTE)
	P.AREA
	P.S.S (P points to a SUBSCR)

8. EXPRESSIONS

Expressions are rules which specify how new values are computed from existing ones. These new values are obtained by performing the operations indicated by the operators on the values of the operands. Expressions fall into two classes: basic expressions - those whose values are of some basic type - and structured expressions - those whose values have some structured type. The former we abbreviate simply by the syntactic class <expression> or <expr>, the latter by <struct exp>.

```
Syntax
<exp>              ::= <expression> | <struct exp>
<expr>             ::= <expression>
<POINTER expr>     ::= <expr> "with type POINTER"
<STRING expr>      ::= <expr> "with type STRING"
<BYTE expr>        ::= <expr> "with type BYTE"
<DESCR exp>        ::= <struct exp> "with type DESCRIPTOR"
                         | <POINTER expr> "to a DESCRIPTOR"
<&DDRESS exp>      ::= <exp> "with type &DDRESS"
```

8.1. Function designators

```
Syntax:
<function designator> ::= <identifier>
                          [ ( <<actual parameter> ,list> ) ]
```

Semantics: A function designator defines a value which can be obtained as follows; the identifier must identify a function. The body of this function is copied, modified by the actual parameters, and executed exactly as specified in Section 9.7. The value is the last value assigned to the function identifier during this execution (undefined if none); its type is the type of the function.

Examples: MAX(X**2, Y)

YOUNGESTUNCLE(JAMES)

8.2. Basic expressions

```
Syntax
<primary>          ::= <constant> | <variable> | @ <variable>
                     | <indirect reference>
                     | <function designator>
                     | <specfunc>
                     | ( <expression> )
<factor>           ::= <primary>
                     | <primary> ** <factor>
                     | <unary op> <factor>
<expression>       ::= <factor>
                     | <expr> <mult op> <expr>
```

```
                    | <expr> <add op> <expr>
                    | <expr> <bit op> <expr>
                    | <expr> <relational op> <expr>
                    | <expr> AND <expr>
                    | <expr> OR <expr>

<unary op>          ::= + | - | NOT
<mult op>           ::= * | / | // | REM
<add op>            ::= + | -
<bit op>            ::= BITOR | BITAND | BITEXOR
<relational op>     ::= = | ¬= | "<" | "¬<" | ">" | "¬>"
```

Note that the above syntax is ambiguous. Expressions are evaluated in a left to right manner, using the precedence of operators given in Section 8.2.2.

8.2.1 primaries. The primaries <constant>, <variable>, <indirect reference> and <function designator> have already been discussed. The primary @ <variable> yields a POINTER value which is the address of (a pointer to) the variable. <specfunc> stands for "special function designator". See Section 2.2.

8.2.2 precedence of operators. Expressions are evaluated in a left to right manner, according to the following hierarchy of operator precedences (parentheses may be used to overide them):

```
unary + unary - NOT
**
* / // REM
binary + binary -
BITOR BITAND BITEXOR
= ¬= < ¬< > ¬>
AND
OR
```

8.2.3 conversion of operands. The following table indicates how values are converted from one basic type to another when necessary. Each row I represents the basic type of a value to be converted, while each column J represents the type to be converted to. The table element (I,J) is then a letter of a footnote below which indicates how the conversion is made. A blank element signifies that no automatic conversion is performed.

RESULT: OPERAND	BYTES	HWI	FWI	DEC	FWF	DWF	POINTER	STRING
.....	-	-	-	-	-	-	-	-

BYTES	A	B	C	C	C	C		I
HWI	D	-	E	E	E	E		
FWI	D	G	-	E	E	E		
DEC	E	E	E	-	E	E		
FWF	F	F	F	F	-	E		
DWF	F	F	F	F	F	-		
POINTER							-	
STRING	J							H

A. If the operand type has fewer bytes than the resulting type, leading zero bytes are added; if the operand has more, leading (leftmost) bytes are discarded until the they have the same length.

B. If the operand is BYTE, it is considered to be an unsigned integer. Otherwise the rightmost two bytes of the operand are considered to be a halfword integer without any other conversion (the leftmost bit is the sign).

C. If the operand has 1,2 or 3 bytes, it is considered to be an unsigned integer and is changed to FWI format. Conversion then proceeds with this new operand. If the operand has 4 or more bytes, the rightmost 4 bytes are considered to be a fullword integer without any real conversion being performed. Conversion then proceeds with this new operand.

D. The HWI (FWI) operand is considered to be a sequence of 16 (32) bits - that is, a BYTE2 (BYTE4) value. The sign bit is just another bit in the sequence. Conversion proceeds with this new operand.

E. Normal conversion. Some significance can be lost in the case FWI to FWF and when the operand is DEC.

F. Normal conversion with truncation. If the result is to be BYTES, the operand is first converted to FWI and then to BYTES.

G. The rightmost 2 bytes are considered to be a halfword. If the operand is between -2**15 and 2**15-1, the result has the same arithmetic value as the operand; otherwise not.

H. If the result has fewer characters, use only the leftmost characters of the operand. If the result has more, add blanks to the right of the operand characters.

I. The operand is assumed to be a string value - each byte is a character. Conversion H above is then performed.

J. The operand characters are considered to be BYTES and the whole operand to be a BYTES value; conversion proceeds from there.

8.2.4 arithmetic operators. The following table defines the

arithmetic operators:

OPERATOR	MEANING
+ A	A (identity)
- A	sign inversion
A ** B	exponentiation of A to the power of B
A * B	multiplication
A / B	division
A // B	integer division. Defined by SGN(A*B) * D(ABS(A),ABS(B)) where SGN is defined by HWI PROCEDURE SGN(FWI X); IF X < 0 THEN SGN=-1 ELSE SGN=1 and D is defined by FWI PROCEDURE D(FWI X,Y); IF X < Y THEN D=0 ELSE D=D(X-Y,Y)+1
A REM B	A - (A//B) * B
A + B	addition
A - B	subtraction

With the arithmetic operations, operands of type BYTE, BYTE1, BYTE2 are considered as positive integers, while a BYTE4 operand is a signed integer (the leftmost bit is the sign). Not all basic type values are valid operands of arithmetic operators. Appendix A contains tables which indicate the valid operands, the automatic conversions performed, and the type of the result of each combination of operator and operands.

8.2.5 bits operators. The bits operators are BITOR, BITAND and BITEXOR. They perform bitwise operations on the two operands as follows:

A	B	A BITOR B	A BITAND B	A BITEXOR B
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

See Appendix A for a list of valid operands, automatic conversions performed, and for the type of the resulting operand.

8.2.6 relational operators. The relational operators yield the result TRUE (X'FF) or FALSE (X'00'), depending on whether the relation is true or not.

If the two operands are arithmetic but have different types,

the value with the lowest type precedence (cf Section 5.1.2) will first be converted to the other type.

If the two operands are of type BYTES but have different lengths, leading zero bytes will be added to the shorter one. The values are considered to be positive integers for the comparison.

If one operand is BYTES and the other arithmetic. The BYTES value will first be converted to type FWI and an arithmetic comparison will be performed.

If the two operands have type POINTER the relation must be = or ¬=. The pointers are equal only if they are both zero or if they point at the same record.

If the two operands are string-valued, the comparison is according to the EBCDIC collating sequence. If the lengths of the operands are different, blank characters are appended on the right of the shorter until the lengths are the same.

Only those combination of operands suggested above are allowed.

8.2.7 logical operators. The operators NOT, OR and AND have the following meaning:

```
NOT A          IF A = 0 THEN TRUE ELSE FALSE
A OR B         IF A ¬= 0 THEN TRUE ELSE B ¬= 0
A AND B        IF A = 0 THEN FALSE ELSE B ¬= 0
```

Note that not only the BYTE values X'FF' and X'00', but all basic values except strings may be operands of the logical operators. Zero means FALSE, anything else means TRUE. Note also that the second operand, B, is not always evaluated. Thus, constructions like

```
IF POINTERVARIABLE AND POINTERVARIABLE.COMPONENT = 3 THEN...
```

Are possible, since if POINTERVARIABLE is zero, the reference to COMPONENT will not be made.

8.2.8 catenation. The CAT operator produces a string whose value is the characters or the first string operand followed by those of the second string operand.

8.3. Structure expressions

Syntax:

```
<struct exp>        ::= <old value> | <altered value>
                        | <new value> | <DESCR exp>
<old value>         ::= <destination>
<altered value>     ::= <destination> ( <component specifier> )
<new value>         ::= <structured type> ( <component
```

specifier>)

<component specifier> ::= <<keyword component> ,list>
| <<positional component> ,list>

<keyword component> ::= <component selector> = _
| <component selector> = <exp>
<positional component> ::= <empty>
| <alternate selector> _
| <alternate selector> <exp>
| <alternate selector>
(<<positional component> ,list>)
<alternate selector>::= <empty> | ¬ <integer>

Semantics: A structure expression yields a value having some structured type. There are three ways of writing a structure expression:

1. The value of an <old value> structure expression is just the current value of the destination. The type must of course be structured. No space is allocated for the value.

2. The value of an <altered value> is found as follows. Space is allocated for the new value. The current value of the destination is moved into this space. The components are then altered as indicated by the component specifier (see below) to yield the resulting value. The destination must of course be structured.

3. The value of a <new value> is found as follows. Space is allocated for a value of the structured type. All components are undefined. The components are then altered as indicated by the component specifier to yield the resulting value.

There are two ways of specifying which components are to be altered - through keyword components and positional components.

1. A keyword component consists of a component selector (cf Section 7.2) which selects the component to be altered, followed by an equal sign, followed by an entity to which the component is to be changed. This entity is either

A. The character "_". This indicates that the component is "empty". The meaning of this will become clear when Section 9.2 on assignment statements is read.

B. An <exp>. The <exp> must be assignment compatible with the component selected. It is evaluated and assigned to the component, exactly as in an assignment statement.

The components are altered in the order in which the keyword components appear (left to right).

2. When positional components are used, the order and number of positional components must correspond to the order and number of constituents of the structured type; the Ith positional component indicates what to do with the Ith constituent. The alternate selector indicates which alternate component of the consituent to use; an empty alternate selector indicates the first alternate.

The entities "_" and <exp> appearing in a positional component have the same meaning as in keyword components (see above). In addition to these there are two more ways of specifying what is to be done with the component:

A. If the positional component is empty (not there), the component is not changed.

B. If the positional component has the form

<alternate selector> (<<positional component> ,list>)

then the corresponding component of the structured type must have subconstituents. This new list of positional components is handled exactly in the same way.

The reader may have noticed that with <altered value> and <new value> structure expressions storage must be allocated. Section 9.2 on assignment statements specifies in which cases it is the programmer's responsibility to release this space.

Examples: We use the structured types

```
STRUCTURE SUBSCR (BYTE AREA, BYTE3 OFFSET, POINTER 5);
STRUCTURE D1 (BYTE KIND ALT HWI B,
                    C(BYTE C1, POINTER C2),
                    SUBSCR D);
SUBSCR V1,V2;
D1 V3,V4;
```

The following is an <old value>: V1

The following are equivalent examples of <altered value>s:

```
V3( B = _, C.C1 = 5, C.C2 = 0)
V3(¬2 _, (5,0),)
```

The following are equivalent examples of <new values>

```
D1(D= SUBSCR(0,0,0))
D1(,,SUBSCR(0,0,0))
```

9. STATEMENTS

A statement denotes a unit of action. To execute a statement means to perform this action. Statements are usually executed in sequence, except when a control or pass communication statement causes a change.

Syntax:

```
<statement>          ::= <open statement> | <closed statement>

<open statement>     ::= <label definition> <open statement>
                       | <open iter state>
                       | <open cond state>

<closed statement>::= <empty>
                       | <label definition> <closed statement>
                       | <compound statement>
                       | <assignment statement>
                       | <closed cond state>
                       | <closed iter state>
                       | <case statement>
                       | <control statement>
                       | <procedure call>
                       | <code statement>
                       | <specfunc> | <specproc>

<label definition> ::= <label> :
```

9.1. Compound statements

Syntax:

```
<compound statement> ::= BEGIN <<statement> ;list> END
```

Semantics: As in ALGOL, the compound statement is used to bracket a sequence of statements.

9.2. Assignment statements

Syntax:

```
<assignment statement> ::= <destination> = <exp>
```

Semantics: This statement is executed as follows:

1. The address of the <destination> is calculated, if necessary.

2. The <exp> is evaluated.

3. The result of (2) is converted and stored - according to the rules given in the table below - at the address calculated in (1). Only those combinations of types of the <destination> and

<exp> are valid which are indicated in the table below. Those pairs of destinations and exps which are valid are called assignment compatible.

The following table indicates how values are converted and assigned to a destination. Each row represents a possible type of the destination; each column a possible type of the <exp>. An element is either blank - which means the combination is not legal - or is a letter identifying a footnote which explains how the conversion and assignment takes place.

Type of destination	type of exp				
	bits	arith	pointer	string	structured
bits	A	A		A	C
arith	A	A			
pointer			B		E
string	A			B	
structured	C				D

A. The conversion is as explained in Section 8.2.3.

B. No conversion necessary.

C. The value of the <exp> as it is in memory is stored in the <destination> without any conversion (zero bytes are added to the right of the <exp> if it is too short, or the rightmost bytes are discarded if it is too long).

D. The <exp> and <destination> must have the same structured type. The <exp> is evaluated and assigned to the destination. That is, components of the destination corresponding to "empty" components in the structure expression (cf Section 7.2) remain unchanged, all others are assigned the value of the corresponding structure expression component. Any space allocated in evaluating the structure expression is automatically released.

E. "empty" components become undefined, and the address of the resulting value is stored into the destination. If space was allocated for the evaluated structure expression, it is now the programmers responsibility to release this space when no longer needed (cf Sections 7.3 and 9.10).

Examples:

```
A = B
P = SUBSCR(A.KIND=5)
CONTENT(P) = SUBSCR(A.KIND=5)
P = CONTENT(P) (A.KIND=3,A.AREA=2,A.OFFSET=_)
```

9.3. Conditional statements

Syntax:

```
<open cond state> ::= IF <expression> THEN
                      <closed statement> ELSE <open statement>
                    | IF <expression> THEN <statement>

<closed cond state>
                  ::= IF <expression> THEN <closed statement>
                      ELSE <closed statement>
```

Semantics: These have the same semantics as in ALGOL.

Examples:

```
IF X = Y THEN GO TO L
IF X THEN U=0 ELSE IF Y=0 THEN U=Y
```

9.4. Iterative statements

Syntax: In the following productions, the letter "J" is to be systematically replaced by the word "open" or the word "closed".

```
<J iter state>    ::= FOR <destination> = <expr¹>
                      [ STEP <expr²> ]
                      UNTIL <expr³> DO <J statement>

                    | WHILE <expression> DO <J statement>

                    | FOR <POINTER destination>
                      IN <tord identifier>
                      [FROM <POINTER expr¹> TO <POINTER expr²>]
                      DO <J statement>

<tord identifier> ::= <table identifier>
                    | <dict identifier>
```

Semantics: The default option for <expr²> is 1. The default option for <POINTER expr¹> and <POINTER expr²> is @<tord identifier> (1) and @<tord identifier> (N) respectively, if the table or dict has presently N records.

The statement

```
FOR I = J STEP K UNTIL L DO <statement>
```

where I is a destination and J, K and L are expressions is equivalent to the following sequence of statements;

```
DEST = @I; &C(DEST) = J;
STEPV = K;
```

```
ENDV = L * SGN(STEPV);
AGAIN: IF &C(DEST) * SGN(STEPV) <= ENDV
THEN BEGIN <statement>; &C(DEST)=&C(DEST)+STEPV; GO TO AGAIN
END
```

where DEST is an internal POINTER variable and STEPV and ENDV are internal variables having the same types as K and L respectively.

The statement

```
WHILE <expression> DO <statement>
```

is equivalent to

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

The statement

```
FOR P IN TAB FROM P1 TO PN DO <statement>
```

where P,P1, and PN are pointers and TAB is a table, is executed as follows:

```
        DEST = @ P; ENDV = PN; CONTENT(DEST) = P1;
AGAIN:  IF CONTENT(DEST) ¬= 0
        THEN BEGIN <STATEMENT>;
                   IF CONTENT(DEST) ¬= ENDV
                   THEN BEGIN TALLY(TAB, CONTENT(DEST));
                              GO TO AGAIN
                        END
             END;
```

where DEST and ENDV are pointer variables.

Examples:

```
FOR I = 1 UNTIL B*3 DO A(I) = I
FOR P.X = 10 STEP - 1 UNTIL 1 DO Y(P.Z) = 5
WHILE PA DO BEGIN PA.D=0; PA = PA.P END
FOR P IN SSYMB DO P.KIND = 0;
```

9.5. Case statements

Syntax:

```
<case statement>  ::= CASE <expression> OF <<statement> ;list>
                      ENDCASE
```

Semantics: The expression is evaluated and assigned to an internal variable I of type FWI. If I <= 0 or I > (the number of statements in the list), no action is taken. Otherwise, the Ith

statement in the list is executed. If this statement does not cause control to leave it, control then passes to the point beyond the ENDCASE symbol.

Example:

```
CASE N OF

Q =5;
FOR I = 1 UNTIL N DO A(I)=0;
GO TO LAB;
BEGIN Q = 5; FOR I = 1 UNTIL N DO A(I)=0 END

ENDCASE
```

9.6. Control statements

Syntax:

```
<control statement> ::= <goto op> <label>
                      | RETURN | SYNTAX | COMPLETE
                      | HALT [ ( <integer> ) ]
                      | CALLPASS ( <pass number> )
                      | BEGINPASS ( <pass number> )

<goto op>           ::= GO | GO TO | GOTO
```

Semantics: Execution of a goto statement transfers control to the statement labeled <label>. One cannot jump into or out of a procedure or into the statement of an iterative statement.

The RETURN statement is used only in procedures; it causes the procedure to return to the point from which it was called.

The SYNTAX statement is used only if the pass has a syntax subprogram. It may not be used in procedures. Execution of the statement causes control to return to the syntax subprogram following the last EXEC action executed.

Execution of COMPLETE tells CIL that the program is done. If CGS was used, the object module for the generated program is completed and written out. Execution then stops.

Execution of HALT (<integer>) causes the message " HALT <integer> " to be printed and execution to halt.

Execution of BEGINPASS causes control to transfer to the beginning of pass <pass number>, while execution of CALLPASS transfers control to pass <pass number> at the place where it last executed a BEGINPASS or CALLPASS (if it had never been executed, control goes to the beginning of it). The CALLPASS is thus like a coroutine call.

If the pass being called is in another coreload, that coreload is brought into core. Passes in the previous coreload may not be called again.

9.7 procedure statements

Syntax:

```
<procedure call>  ::= <identifier>
                      [ ( <<actual parameter> ,list> ) ]
                      | <specfunc> | <specproc>

<actual parameter>::= <expression> | <table identifier>
                      | <dict identifier> | <stack identifier>
```

Semantics: Execution of a procedure statement is equivalent to the following process:

A copy is made of the procedure or function body identified by the identifier in the procedure statement. The actual parameters of the procedure statement, which must agree in number and order with the formal parameters of the procedure or function, systematically replace those formal parameters as follows:

1. If the actual parameter is a <destination> whose type is the same as the type of the formal parameter, the address of the <destination> is calculated and assigned to an internal variable, say I, which is different from any other variable. The indirect reference "&C(I)" then replaces every occurrence of the formal parameter identifier in the copy of the procedure body.

2. If the actual parameter is a constant, the constant is converted to the type of the corresponding formal parameter (this must be possible) if necessary and the result replaces every occurrence of the formal parameter.

3. If the actual parameter is any basic expression not covered in 1 or 2, it is evaluated, assigned to an internal variable, say J, whose type is the same as the type of the corresponding formal parameter. The variable J then replaces every occurence of the formal parameter.

4. If the actual parameter is a table, dict or stack identifier, the corresponding formal parameter must be a table, dict or stack, respectively, with the same type. The actual parameter replaces every occurrence of the formal parameter identifier in the copy of the procedure body.

The replacement of parameters must yield valid expressions and statements. The modified copy of the procedure body is then

executed.

If a function is executed in this manner, the value it produces is lost.

<specfunc>s and <specproc>s are calls on special functions and special procedures. See Section 2.2.

Examples:
£JOINREGS(P)

TIME
LOOKLAB(A, PP)
YOUNGESTUNCLE(JOHN)

9.8 scanner statements

The following <specproc>s are used to communicate with the scanner:

SCAN
CHARMODE
NORMODE
SCANNER (<scanner id>)

Execution of SCAN causes the next symbol to be read from the source language program being compiled. It is put in location SCANSYM and on the main stack of the pass in which the SCAN appears (if applicable). See Section 12.1 for an exact description.

Execution of the statement CHARMODE causes the scanner to change its method of scanning the source program to a character by character scan. See Section 12.1.

Execution of the statement NORMODE causes the scanner to scan the source program in normal fashion. See Section 12.1.

Execution of SCANNER (<scanner id>) causes the scanner to begin using the scanner definition named <scanner id> for forming source language symbols.

9.9. Input-output

The I/O provided is quite primitive. More powerful I/O may be added at a later date if necessary.

9.9.1. Input. Section 12.6 explains input procedures when the normal scanning is performed (cf Section 9.8). In addition, the <specproc>

```
| &IN |
```

reads the next card into the system string variable &INLINE.

9.9.2. Output. Execution of the <specproc>

```
| &OUT ( <<expr> ,list> ) |
```

causes the expressions to be added to the current output line. Strings are added without conversion. Pointer and bits type expressions are first converted using the function &HEXT (see below), HWI, FWI and DEC expressions are first converted using the function &DECT, while FWF and DWF expressions are first converted using the function &FLPT. When the current output line is filled up a new one is started, execution of the <specproc>

```
| &OUT |
```

causes the current line to be written out (if not empty).

Execution of the <specproc>

```
| &OUTDESCR ( <DESCR exp> ) |
```

causes the current output line to be written out and the DESCRIPTOR to be written out in a readable form.

9.9.3. Conversion functions. The following <specfunc>s return a binary representation of the STRING parameter S:

<specfunc>	S is	S can contain only the characters	result is
&TBIN(S)	binary	0,1	BYTES
&TOCT(S)	octal	0,...,7	BYTES
&TDEC(S)	decimal	0,...,9	FWI

&THEX(S)	hexadec.	0,...,9,A,...,F	BYTES

The result is right-adjusted with leading zeroes if necessary. The number of bytes is the number necessary to represent the string in binary. An error message is printed if S contains illegal characters.

The following <specfunc>s perform the same function except that the parameter A is an atom (BYTE2 representation) of the string S:

&TBIN(A)
&TOCT(A)
&TDEC(A)
&THEX(A)

The following <specfunc>s are used to convert an internal number to character form. The result is thus a STRING expression. Below, A represents an atom (BYTE2 expression).

<specfunc>	the STRING result is
&BINT(<expr>)	<expr> expressed in binary characters
&DECT(<expr>)	<expr> expressed in decimal char.
&FLPT(<expr>)	<expr> expressed in floating pt. Char.
&HEXT(<expr>)	<expr> expressed in hex characters.
&OCTT(<expr>)	<expr> expressed in octal characters.
&TEXT(A)	string corresponding to atom A

No conversion is performed on <expr>; it is changed as it stands in memory.

Examples:

```
&BINT( B'11010') is equal to '11010'
&DECT( B'11010') is equal to '26'
&FLPT( B'11010') is equal to '2.6 E+01'
&HEXT( B'11010') is equal to '1A'
&OCTT( B'11010') is equal to '32'
&DECT( -3645001) is equal to '-3645001'
```

9.10. Releasing storage

If an assignment statement

<POINTER destination> = <struct exp>

(where the <struct exp> is not an <old value>) is executed, CIL allocates storage for the <struct exp> and puts its address in the <POINTER destination>. It is then the programmers responsibility to release this storage when no longer needed (see Section 14.9 for the special case of DESCRIPTORS). The <specproc>

```
| &RELEASE ( <POINTER destination> [,<pointo type> ] ) |
```

releases the storage pointed at by the POINTER and sets it to zero. The <pointo type> is needed if the declaration of the POINTER did not unambigously indicate the data being pointed at.

10. OPERATIONS ON TABLES, DICTS AND STACKS

This section describes how one adds, deletes and searches for records in tables, dicts and stacks. Each <specfunc> described here yields a POINTER value - either 0 or the address of a table, dict or stack record. Thus they may be used anywhere a function designator is used. They may also appear separately like procedure statements, in which case their value is lost.

10.1. Operations on tables

Syntax: The syntax of the ENTER, LOOK, TALLY and DELETE <specfunc>s is

```
ENTER ( <table identifier> [ , <exp> ] )

LOOK ( <table identifier> [ . <component selector> ]
       , <expression>
       [ FROM <POINTER expression¹> ]
       [ TO <POINTER expression²> ]
       [ , BACK ] )

TALLY ( <table identifier> , <POINTER expression> )
       [ , BACK ] )

DELETE ( <table identifier> , <POINTER expression>)
```

Semantics:

ENTER. A new record is added to the table identified. If the <exp> is present <exp> (which must be assignment compatible with the type of the table records) is assigned to this new record; otherwise its value is undefined. The value of ENTER is the address of the new record.

LOOK. If the type of the records of the table is a basic type, the component selector may not appear. A subset of the records is searched for one which is equal to <expression>. If the type of the records is a structured type, a subset of the records is searched for one whose component selected by the component selector (default option is "1¬1") is equal to <expression>. The comparison is done according to the rules of Section 8.2.6.

<POINTER expression¹> must point at a record of the table, say the Ith (default option is the address of the first record). <POINTER expression²> must point at a record of the table, say the Jth (default option is the address of the last record).

If BACK is missing, the records tested are records I,

I+1,...,J, in that order (none if J < I). If BACK appears, records J,J-1,...,I are tested, in that order (none if J < I).

If a record is found, the value of LOOK is the address of the record. Otherwise the value is 0.

TALLY. The POINTER expression must be 0 or the address of a record of the table identified. The value of TALLY has type POINTER and is given by the following table, assuming the table has N records.

POINTER expression	Value if BACK is not present	Value if BACK is present
0	addr. Of record 1	addr. Of record N
addr. Of record 1	addr. Of record 2	0
addr. Of record N	0	addr. Of record N-1
addr. Of record J (1 < J < N)	addr. Of record J+1	addr. Of record J-1

DELETE. The POINTER expression must be the address of a record in the table, say record I. If there are currently N records in the table, records I, I+1, ... , N are deleted from the table. The value of DELETE is the value of the new last record - record I-1 (0 if table is now empty).

10.2. Operations on dicts

Syntax: The syntax of these <specfunc>s ENTER, LOOK, TALLY and DELETE is

```
+-------------------------------------------------------+
| ENTER ( <dict identifier> , <BYTE2 expression>        |
|         [ , <exp> ] )                                 |
|-------------------------------------------------------|
| ENTER ( <dict identifier> , <POINTER expression>      |
|         [ , <exp> ] )                                 |
|-------------------------------------------------------|
| LOOK ( <dict identifier> , <BYTE2 expression> )       |
|-------------------------------------------------------|
| LOOK ( <dict identifier> , <POINTER expression> )     |
|-------------------------------------------------------|
| TALLY ( <dict identifier> , <POINTER expression>      |
|         [ , BACK ] )                                  |
|-------------------------------------------------------|
| DELETE ( <dict identifier> , <POINTER expression> )   |
+-------------------------------------------------------+
```

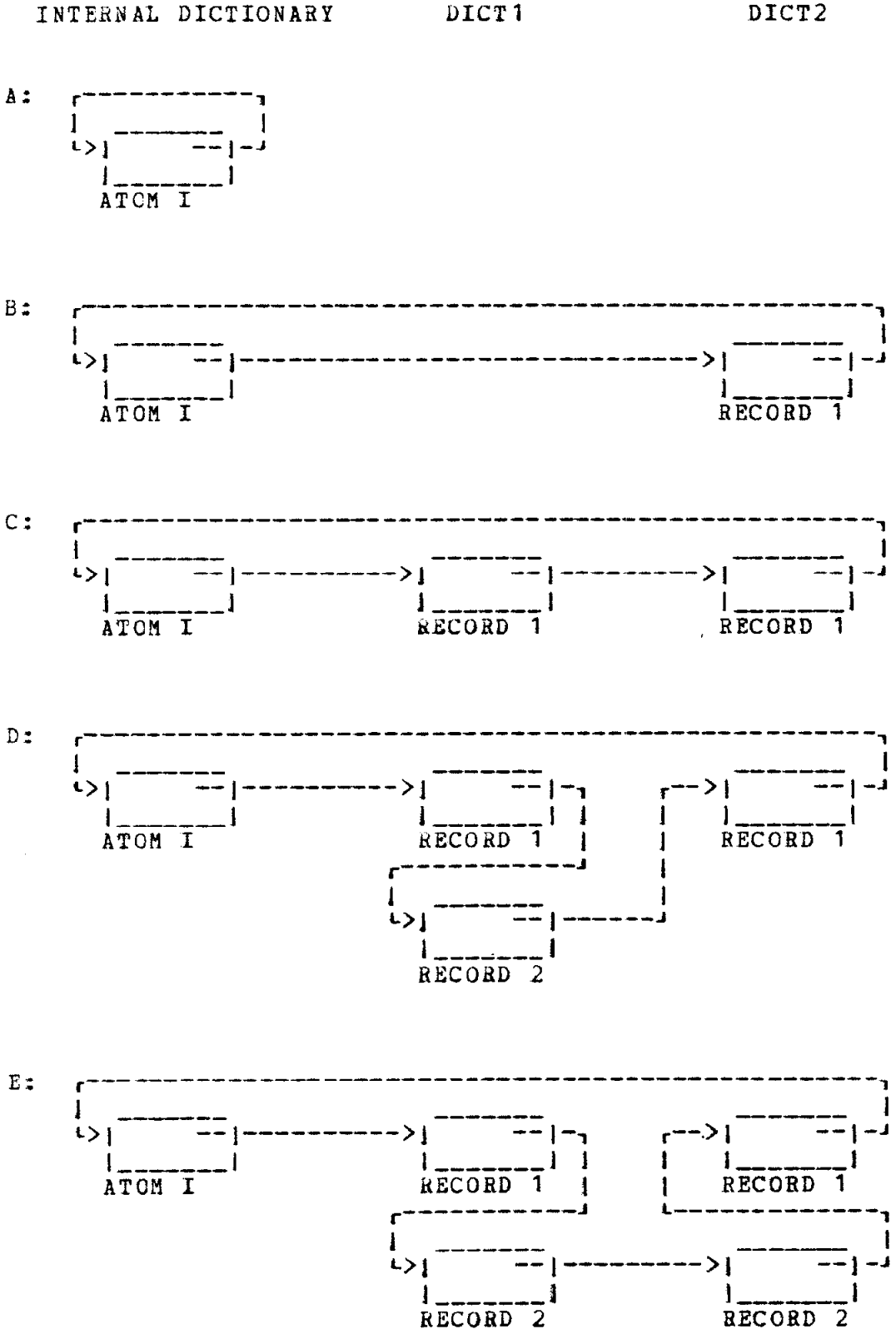
Semantics: As discussed in Section 12 on the scanner

definition, each compiler automatically uses a hash-coded internal dictionary &INTDIC to aid in changing from source language symbols to their internal representations called ATOMs. There is one record in the internal dictionary for each source language symbol recognized. By using dicts the compiler writer can use the internal dictionary to search his own symbol tables efficiently.

In the discussion of dict declarations it was stated that the structured type of the records must begin with a BYTE component followed by a POINTER component. The first component automatically contains an internal number identifying the dict. The second component is used to chain dict records which refer to the same ATOM to the internal dictionary record for that ATOM. Thus, in order to find the record in a dict for an identifier, one only has to search the chain based on the internal dictionary record for that identifier.

Fig. 1, part A shows the record for an ATOM, I, before any dict records have been chained to it; the second component of the record points to the record. In the same part A it is assumed that the dicts DICT1 and DICT2 are empty; the other parts of figure 1 will be used to illustrate the operations on dicts.

FIGURE 1


INTERNAL DICTIONARY
DICT1
DICT2
A:
ATOM I
B:
ATOM I
RECORD 1
C:
ATOM I
RECORD 1
RECORD 1
D:
ATOM I
RECORD 1
RECORD 1
RECORD 2
E:
ATOM I
RECORD 1
RECORD 1
RECORD 2
RECORD 2

ENTER. A new record is added to the dict identified. If <exp> is present, it is assigned to the record (it must of course be assignment compatible with the record); otherwise the record value is undefined. The record is then chained to the internal dictionary, as follows:

1. If the second parameter is a BYTE2 expression, its value must be an ATOM - that is, the internal representation of some source language symbol. The new record is inserted in the chain directly after the internal dictionary record for the atom. As an example, consider fig. 1, part A. Executing

```
ENTER(DICT2,I)
```

would yield fig. 1, part B. Further execution of

```
ENTER(DICT1,I)
```

would yield fig. 1, part C.

2. If the second parameter is a POINTER expression, its value must be the address of some chained dict record (not necessarily the dict identified in the ENTER operation.) the new record is inserted in the chain after the chained dict record. For example, consider fig. 1, part C. If P is a POINTER variable, executing

```
P = ENTER( DICT1, DICT1(1))
```

would yield part D. Further execution of

```
ENTER(DICT2, P)
```

would yield part E.

LOOK. There are two variations:

1. If the second parameter is a BYTE2 expression, its value must be an ATOM. The chain based on that ATOM is searched for a record in the dict. The value of LOOK is the address of the first one found (0 if none found). For example, consider fig. 1 part D. Execution of

```
LOOK(DICT2,I)
```

yields the the address of the record DICT2(1), while execution of the same statment but with the configuration of fig. 1 part E would yield the address of DICT2(2).

2. If the second parameter is a POINTER expression, its value must be the address of some chained dict record. The records after the one addressed and up to the internal dictionary record are searched for one in the dict specified. The value of LOOK is the address of the first one found (0 if

none found). For example, consider fig. 1 part E. Executing

LOOK(DICT1, DICT1(1))

yields the address of DICT1(2), while executing

LOOK(DICT1, DICT1(2)) or LOOK(DICT1, DICT2(2))

yields the value 0.

TALLY. This works exactly as the TALLY operation with tables.

DELETE. This works exactly as the DELETE operation with tables, with the addition that the records are taken off the chain before being deleted. For example, consider fig. 1 part E. Execution of

P = DELETE(DICT2, DICT2(2))

yields the address of record DICT2(1) in P and the configuration in fig. 1 part D.

10.3. Operations on stacks

Syntax: The form of the PUSH and POP <specfunc>s is

```
┌-------------------------------------------------┐
| PUSH ( <stack identifier> [ , <exp> ] )         |
|-------------------------------------------------|
| POP ( <stack identifier> [ , <destination> ] )  |
└-------------------------------------------------┘
```

Semantics:

PUSH. Executing PUSH adds a new record to the stack identified. The value of the record is the value of <exp> (which must be assignment compatible with the record), if present; otherwise it is undefined. The value of PUSH is the address of the new record.

POP. Executing POP deletes the top record from the stack identified. If the destination is present, the top record (which must be assignment compatible with the destination) is first assigned to the destination. The value of POP is the address of the new top stack record (0 if the stack is now empty).

Care must be taken when PUSHing and POPing the main stack of a pass; a semantic routine should not PUSH and POP if it later refers to the main stack via L0, L1, L2, L3, L4, L5, R1, R2, or R3.

10.4. The table &INTDIC.

&INTDIC is the hash-coded INTernal Dictionary used to transform source language symbols into atoms. The following <specfunc>s are provided to allow a compiler writer some access to it.

LOOK (&INTDIC, <STRING expr>)
ENTER(&INTDIC, <STRING expr> [,<BYTE expr>])
ATOM (<POINTER expr>)
ATOM (<STRING expr>)
&TYPE(<POINTER expr>)
&TYPE(<STRING expr>)

LOOK returns the address of the &INTDIC record for the STRING expression (or 0 if no record for it).

ENTER is executed as follows: If no record exists for the STRING, one is added to &INTDIC. Then the value of the BYTE expression becomes the type of the string for the current scanner definition (cf Section 12.1). The default option for the BYTE expression is 0. The value of this <specfunc> is the address of the &INTDIC record.

ATOM returns a BYTE2 value. In the first case, the POINTER expression must yield the address of an &INTDIC record or a dict record. The value returned is the atom for the symbol associated with the record. In the second case, the value returned is the value assigned to the BYTE2 variable B when the following statements are executed:

```
P = LOOK(&INTDIC, <STRING expr>);
IF P
THEN B = ATOM(P)
ELSE B = ATOM( ENTER(&INTDIC, <STRING expr>) );
```

&TYPE returns a BYTE value - the type of the symbol (cf Section 12.1) associated with the &INTDIC or dict record pointed at by the POINTER expression (case 1) or with the STRING expression (case 2) - which must already be in &INTDIC.

11. STORAGE ALLOCATION AND ALIGNMENT OF VALUES

While not necessary, it is often helpful to know how storage is allocated. In the IBM 360, data must often begin on a halfword, fullword or doubleword boundary. We define the alignment factor as follows:

data must begin on	alignment factor is
doubleword	8
fullword	4
halfword	2
byte	1

In other words, if the alignment factor is i then the address of the leftmost byte of the data must be a multiple of i . The following table gives the alignment factor and storage requirement for basic type values.

Type	alignment factor	number of bytes used
BYTE	1	1
BYTE2	2	2
BYTE3	4 (see A below)	(see A below)
BYTE4	4	4
BYTES(I)	1 (see B below)	I
HWI	2	2
FWI	4	4
DEC	to be determined later	
FWF	4	4
DWF	8	8
POINTER	4 (see A below)	(see A below)
STRING(I)	1 (see B below)	I

A. BYTE3 and POINTER values are contained in the last 3 bytes of an IBM 360 fullword. The first byte may or may not be used for another value.

B. In certain cases, a BYTES or STRING variable may be given four bytes - one for the length minus 1 and the other three for the address where the actual value really is.

The following rules are used to allocate storage for structured type values.

1. The alignment factor for a structured type value is the maximum of the alignment factors of all its components and subcomponents.

2. The alignment factor for any component with subcomponents is the maximum of the alignment factors of those subcomponents.

3. All components of a constituent use the same space. The number of bytes allocated to a constituent is the maximum of the number of bytes needed for each component of the constituent.

4. Bytes are allocated to constituents in a left-to-right manner. The constituents are packed together as close as possible, taking into account their alignment factors.

5. The number of bytes used is a multiple of the alignment factor.

Examples

```
STRUCTURE S1(BYTE B, HWI C, BYTE D1 ALT HWI D2);

STRUCTURE S2(BYTE E, FWI F, G(BYTE2 G1 ALT BYTE G2), S1 H);

S1 A1 ; S2 A2;
```

A1 will begin on a halfword, A2 on a fullword. They look as follows (bytes are numbered starting at 0; the underlining after each identifier indicates which bytes that component uses):

```
BYTE    0  1  2  3  4  5
        A1_______________
        B__   C_____D1_
                    D2____

BYTE    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
        A2_____________________________________________________
        E__         F__________G_____H_______________
                               G1____B__   C_____D1_
                               G2_             D2____
```

12. SCANNER DEFINITIONS

The scanner is that part of a compiler which reads in the original source program characters and composes them into atoms - identifiers, integer, single and double character delimiters, and reserved words. The scanner definition indicates how these atoms are to be formed.

As indicated in Section 4, several scanner definitions may be given. Initially, the first one is in effect until changed at compile time.

The scanner definition was defined with two conflicting goals in mind:

1. The scanner should be efficient. To accomplish this, the IBM 360 "translate and test" instructions are used, along with three or four 256-byte tables per scanner definition. With this, for example, sequences of 1 to 256 blanks in the input source program can be skipped with one instruction.

2. The scanner definition should be flexible enough to accomodate all existing languages. This of course was not possible. In order to accomodate more languages, the compiler writer can test, insert and delete characters from each card before it is actually scanned. He can also switch back and forth from normal scanning to character - by - character scanning (in which case he builds atoms himself).

At this point, an example will help to make the next sections easier to understand. Suppose our source language consists of octal expressions using the operators +,-,*,/ and **. Parentheses (and) are also used. Numbers are octal integers. Identifiers must begin with $ or one of the letters A through J; the succeeding characters must be one of the letters A through J. IDBEG is a reserved word used to identify the beginning and end of expressions. Comments begin with /* and end with */. Spaces are ignored. The scanner definition is

```
SCANNER ONE                  (ONE identifies the scanner def)
SYN IDBEGSYN IDBEG           (IDBEGSYN is a synonym for IDBEG)
DIGIT 0 1 2 3 4 5 6 7        (defines digits)
IDBEG $ A B C D
E F G H I J                  (defines beginning id chars.)
IDCHAR A B C D E F G H I J   (defines other id chars.)
TERMIN + - * / ( )           (defines single
                             character delimiters)
IGNORE X'40'                 (spaces are completely ignored)
INVTERMIN NONE               (this class of symbols is empty)
RES IDBEGSYN **              (defines reserved words and
                             2-character delimiters)
COMMENTQ /* */               (comments begin with /*
                             and end with */)
ENDSCAN                      (end of scanner definition)
```

Syntax:

```
<scanner def>   ::= SCANNER <scanner id>
                    [ <<synonym def> list> ]
                    <<set definition> list>
                    [ <<reserved def> list> ]
                    [ <<quote def> list> ]
                    [ BEGIN <preprocessor> ]
                    ENDSCAN
```

12.1. Scanning and the internal dictionary

When scanning a source program, the scanner proceeds from left to right through the program. The end of a line (card) has no significance. (the compiler writer may, however, have his own internal character inserted at the end of each line to give it some significance - cf Section 12.6). In case there are several alternatives for the next source language symbol, the scanner always picks the longest one. Thus if 'BEGIN' and 'BEGIN are both reserved words and the characters ', B, E, G, I, N and ' are scanned, then 'BEGIN' will be formed.

Scanning in normal mode (NORMODE) (cf Section 9.8). When a source language symbol is formed, it is replaced by a 16-bit number. The compiler works exclusively with this number. The word atom is used both for a source language symbol and its 16-bit representation.

In order to replace a symbol by its 16-bit representation, the system uses a hash-coded internal dictionary, named &INTDIC. &INTDIC contains a record for each source language symbol scanned. Besides the symbol itself and its internal representation, this record indicates (for each scanner definition) how the symbol has been used. The possibilities are:

type	meaning
0	The symbol is undefined (has not been scanned using this scanner definition).
6	The symbol is an identifier (I).
7	The symbol is a number (N).
8	The symbol is a string (S).
9	The symbol is a reserved word or terminator (like + - BEGIN END) (R).
10	The symbol begins a comment (CQ).
11	The symbol begins a string (SQ).

When an atom is scanned, it is passed to the compiler in location SCANSYM. SCANSYM contains two BYTE2 components. Just how the atom is put in SCANSYM depends on its use. If it is a reserved word or terminator (R), the atom for it is put in SCANSYM.1 (first component), while SCANSYM.2 becomes undefined. If it is an identifier (number or string), the metasymbol I (N or S) is put in

SCANSYM.1 and the atom for the identifier (number or string) itself is put in SCANSYM.2.

Scanning in character mode (CHARMODE) (cf Section 9.8). When in character mode, the source program characters are put in SCANSYM.1 as they are scanned. SCANSYM.2 becomes undefined.

12.2. Defining synonyms

Syntax:

```
<synonym def>    ::= SYN <<synonym pair> list>
<synonym pair>   ::= <synonym> <<EBCDIC char> list>
                   | <synonym> <char sequence>

<char sequence>  ::= <EBCDIC or hex>
                   | <char sequence> CAT <EBCDIC or hex>
<EBCDIC or hex>  ::= <EBCDIC char> | <hex char>
<hex char>       ::= X ' <hexit> <hexit> '
```

Semantics: A <hex char> may not be X'70'. The <hex char> allows one to use other 8-bit combinations as characters, besides the EBCDIC bit combinations. Note that a space must be represented by its hex representation, X'40'.

The synonym definition associates a CIL identifier (the synonym) with a sequence of characters which form a source language symbol (the <EBCDIC char> list or the <EBCDIC or hex>s in the <char sequence>). The synonym must be used later in a set definition (cf Section 12.3) or in a reserved word definition (cf Section 12.4), to indicate how the source language symbol is used.

Any source language symbol can be given a synonym; the following must have a synonym:

1. Those source language symbols which are scanner definition reserved words:

```
BEGIN
CAT COMMENTQ
DIGIT
ENDSCAN
IDBEG IDCHAR IGNORE INVTERMIN
NONE
RES
STRINQ SYN
TERMIN
```

2. Those source language symbols which contain (or are) a space or a character which is not an EBCDIC character.

A synonym may not be a reserved word of a sublanguage in which it is

used (production language or semantic sublanguage or scanner definition.)

12.3. Set definitions

Syntax:

```
<set definition> ::= DIGIT <char set>
                   | IDBEG <char set>
                   | IDCHAR <char set>
                   | TERMIN <char set>
                   | INVTERMIN <char set>
                   | IGNORE <char set>

<char set>       ::= NONE | <<character> list>
<character>      ::= <EBCDIC or hex> | <synonym>
```

Semantics: Set definitions serve to describe the use of each character in the source language. Each character must appear in at least one set definition. These definitions are used by the scanner to build the actual source language symbols. A set definition with the <char set> NONE defines an empty set. The sets have the following meaning:

1. The set of DIGITs are used to form numbers according to the syntax

<source number> ::= <<digit> list>.

When a source number is formed, the metasymbol N is returned in SCANYSM.1, while the atom for the source number itself is put in SCANSYM.2. Note that no actual conversion of the number is performed.

2. The sets IDBEG and IDCHAR are used to form source (language) identifiers according to the syntax

<source id> ::= <char in set IDBEG>
[<<char in set IDCHAR> list>].

When a source identifier is formed, the metasymbol I is returned in SCANYSM.1, while the atom for the source identifier itself is put in SCANSYM.2.

3. The set TERMIN contains the single character symbols of the source language. Examples from ALGOL and FORTRAN are + - (and) . These characters are called terminators, since they terminate identifiers or numbers. When scanned, the atom for a terminator is put in SCANSYM.1 while SCANSYM.2 becomes undefined.

4. The characters in the set INVTERMIN signal the end of an atom being formed. For example, in some languages a space

following an identifier ends that identifier; **A B** is two identifiers - A and B. However, these characters are **INVisible** - they are not passed on to the compiler (except in strings).

5. The characters in the set IGNORE are completely ignored (except in strings) if they appear in the source program. For example, in some ALGOL implementations blanks are ignored; A BC is the identifier ABC.

The default option, in case a set definition for one of the sets is missing, is taken from the following set definitions:

```
DIGIT 0 1 2 3 4 5 6 7 8 9
IDBEG 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
IDCHAR  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 0 1
2 3 4 5 6 7 8 9
TERMIN NONE
INVTERMIN X'40'
IGNORE NONE
```

The following restrictions are placed on the sets. The sets IDBEG and IDCHAR may have a nonempty intersection. The sets IDCHAR and DIGIT may have a nonempty intersection. The intersection of any other two sets must be empty.

12.4. Reserved words

```
Syntax
<reserved def>  ::= RES <<res word> list>
<res word>      ::= <source id>
                  | <termin> <source id> [ <termin> ]
                  | <termin> <termin>
                  | <synonym>
<termin>        ::= "a character in the set TERMIN"
<source id>     ::= "a source language identifier (cf sets
                    IDBEG, IDCHAR)"
```

Semantics: The reserved definition declares the reserved words of the source language. Note that we include double character symbols like // and /* here. If a synonym appears here, the source language symbol it represents must have one of the other forms given above.

12.5. String and comment quotes

```
Syntax
<quote def>     ::= STRINGQ <<quote pair> list>
                  | COMMENTQ <<quote pair> list>

<quote pair>    ::= <begin quote> <end quote>
<begin quote>   ::= <termin> | <res word> | <synonym>
<end quote>     ::= <termin> [ <termin> ] | <synonym>
```

Semantics: The set COMMENTQ contains pairs of beginning and end quotes for comments. The beginning quote can be any terminator or reserved word; the end quote must consist of one or two terminators. Comments are deleted from the source program. A comment is thus an invisible terminator (set INVTERMIN).

The set STRINGQ contains pairs of beginning and end quotes for strings. The beginning quote can be any terminator or reserved word, while the end quote must consist of one or two terminators. When a string is detected, the metasymbol S is put in SCANYSM.1 and the atom for the string (without the quotes) is placed in SCANSYM.2.

12.6. Processing before scanning

Syntax: <preprocessor> ::= <procedure call> " of a procedure without parameters"

Semantics: The procedure must be in core during the time the scanner definition is used. When reading in a new source program line, the scanner puts it in the system string variable &INLINE and executes the procedure call. This procedure can then do any preprocessing necessary before the scanner actually scans the line. The result of this preprocessing must be put in the system string variable &SCLINE. The original line should also be written out using &OUT.

For example, suppose we wish to preprocess a FORTRAN program. The end of a line means the end of a statement except when column 6 of the next card is nonblank. In addition, columns 1-5,7-72 are fixed fields. Suppose in the scanner definition we declared the terminals EOS (end of statement) and EOL to be two byte representations which cannot appear on the input card. The following procedure then will accomplish what we want:

```
PROCEDURE PREPROC;
  BEGIN &OUT(&INLINE); &OUT;     /* write out the line */
   IF SUBBYTE(&INLINE,5,1) = ' '
   THEN BEGIN                   /* this is not a continuation card*/
        &SCLINE = EOS           /* put in end of statement,*/
                  CAT SUBBYTE(&INLINE,0,5) /* label field,*/
                  CAT EOL                  /* end of label,*/
                  CAT SUBBYTE(&INLINE,6,66) /* rest of card */
        END
   ELSE BEGIN                   /* continuation card. */
        &SCLINE = SUBBYTE(&INLINE,6,66)
        END
  END
```

If the <preprocessor> is missing from a scanner definition, a procedure with the following procedure body is automatically invoked before each new line is scanned:

```
BEGIN &OUT( &CLINE, '  ',          /* line number */
            &INLINE);              /* input line  */
   &OUT;
   &SCLINE = SUBBYTE(&INLINE,0,72); /* only cols 1 to 72 */
END
```

The following system indentifiers are used in connection with the scanner.

STRING(80) &INLINE. Always contains the last source program line.

STRING(256) &SCLINE. Current source program line being worked on.

STRING(5) &CLINE. Contains the number of the current line (with leading blanks).

HWI &NLINE. The number of the current line.

13. PRODUCTION LANGUAGE (PL)

Production language (PL) is a sublanguage of CIL designed for writing "parsers" or "syntax analyzers" of programs. It consists primarily of so-called productions which work with a LIFO stack. Briefly, as a source program is scanned, the source symbols are placed on the stack and an attempt is made to match the top stack symbols with those designated by the current production. If no match occurs the following production becomes the current one and a match is attempted again; this continues until a match occurs. When it does, the top of the stack is rearranged and several actions are performed as indicated by the current production. These actions may cause more symbols to be stacked, may cause a portion of the semantic subprogram to be executed and may also indicate which production is to become the current production.

At this point an example might help to make this whole section clearer. Consider the following production:

```
IF E THEN       > ICL     EXEC SIFCLAUSE  GO THENPART
```

This production has the following meaning: If the top three stack records contain the symbols IF, E and THEN, then replace these three records by a single record containing the symbol ICL, execute that portion of the semantic subprogram labeled SIFCLAUSE, make the production labeled THENPART the current production and begin matching again.

Production language need not be used, in which case the semantics portion of a pass is executed as a program in the usual manner; statements are executed in the order in which they appear.

If production language is used in a pass, then it is the production language subprogram which is in command - which drives the compiler. It causes source language symbols to be scanned and invokes parts of the semantic sublanguage.

Syntax:

```
<PL subprogram>   ::= PRODLANG
                      [ <<PL declaration> list> ]
                      PRODUCTIONS <<production> list>
                      ENDSYNTAX
```

13.1 comments and blanks

A comment in PL is any sequence of characters, not including the subsequence "::", enclosed in the comment quotes "::" and "::" . A comment may appear between declarations and/or productions.

Blanks may appear anywhere except between characters of a <PL identifier>, <source symbol>, <identifier>, or reserved word. At least one blank must separate them if they are adjacent.

13.2 PL reserved words

The reserved words of production language are:

```
ANY
CALL CLASS CLASSLAB
ENDSYNTAX ERROR EXEC
GO
HALT
I IF INT
N
PRODLANG PRODUCTIONS
RETURN
S SCAN SCANNER SCANSYM SIGNAL STAK
UNSTK
```

They may not be used as identifiers in a PL subprogram.

13.3. Source language symbols

Syntax

<source symbol> ::= "any sequence of 1 to 250 EBCDIC characters except "$", ":", ">" and space(blank). It may not be a PL reserved word."
| <source language symbol>

Semantics: A source symbol is a sequence of characters which was declared in a scanner definition to be a symbol of the language to be compiled (cf Section 12.2). Note that in this subprogram only, the source symbol may appear without the "$" in front of it, as long as it follows the rules given above.

Examples:

```
BEGIN
+
$$ is the source language symbol "$"
CLASS is not a source symbol since it is a reserved word.
$CLASS represents the source language symbol CLASS.
AND and $AND are equivalent.
```

13.4. Metasymbols

Syntax: <metasymbol> ::= I | N | S | ANY

The metasymbols I, N, and S represent an identifier, a number (sequence of digits) and a string of the source language being compiled, respectively. ANY represents any source language symbol. Their use will be explained later.

13.5 PL identifiers

Syntax

```
<PL identifier>   ::= "any sequence of 1 to 250 EBCDIC
                      characters except "$", ":", ">", and
                      space. It may not be a PL reserved word
                      or be used as a <source symbol>."
<PL label>        ::= <PL identifier>
<PL int>          ::= <PL identifier>
<class name>      ::= <PL identifier>
```

Semantics: By the symbol <identifier> we mean the usual identifier (cf Section 1.3) - a sequence of letters and digits, the first of which must be a letter. <identifier>s used in a PL subprogram are declared elsewhere - as a synonym for a source language symbol, as a label in the semantic program, etc.

PL identifiers - those declared and used only in a PL subprogram - are less restricted, as indicated by the above syntax. A PL identifier may be declared only once in a PL program and must be different from any identifier or symbol used in a PL subprogram.

13.6. Communication between syntax and semantics

13.6.1 the main stack

Production language uses a LIFO stack. This stack serves also as the major communication between the production language subprogram and the semantic subprogram. The stack to be used for this purpose is defined by a <main stack dec> in the semantic sublanguage (cf Section 6.2). It must be STATIC (cf Section 6.2) and the first three components of the stack records must be of type BYTE2. Apart from this, the compiler writer is free to define the structure of the stack record as he chooses. The second component is called the syntax component of the stack; it is used to store the (atoms for the) symbols of the language.

As source language symbols are scanned at compiletime, they are put in location SCANSYM (cf Section 12.1) and then pushed onto the stack as follows:

1. If the symbol is an identifier (number or string), the metasymbol I (N or S) is put into the second BYTE2 component, and the atom for the identifier (number or string) is put into the third BYTE2 component. The first component is reserved for system use.

2. If the symbol is not an identifier (number or string), its atom is put in the second BYTE2 component. The first component is reserved for system use while the third component becomes undefined.

For example, suppose the string "A = B PLUS 1" is scanned, where A and B are identifiers, PLUS is a reserved word and 1 is a number in the source language being compiled. Then the stack would be:

stack rec.	1st comp.	2nd comp.	3rd comp.
0	reserved	N	1
1	reserved	PLUS	undefined
2	reserved	I	B
3	reserved	=	undefined
4	reserved	I	A

13.6.2 location SIGNAL

BYTE identifier SIGNAL is a system identifier local to a pass whose value can be changed in the usual manner in the semantic sublanguage and tested in production language. Its value is initially undefined. (cf Section 13.9, action 7).

13.7. Declarations in PL

Syntax

```
<PL declaration>  ::= <int dec> | <class dec>
                      | <classlab dec>

<int dec>         ::= INT <<PL int> list>
<class dec>       :== CLASS <class name> <<symbol> list>
<classlab dec>    ::= CLASSLAB <class name>
                      <<symbol-label> list>

<symbol label>    ::= <symbol> <label>
<symbol>          ::= <source symbol> | <PL int> | I | N | S
                      | <int identifier>
```

Semantics: The identifiers declared in an INTernal declaration can be thought of as "nonterminal" symbols used to help define the syntax of the source language. They can be placed in the syntax portion (second component) of the stack. Each INT identifier is represented internally by a 16 bit (BYTE2) integer assigned by CIL.

CLASS and CLASSLAB declarations serve to associate the <symbol>s with the class name. This is simply a notational convenience; a production containing a class name is equivalent to a sequence of productions, each with one of the <symbol>s substituted for the class name.

Additionally, a CLASSLAB declaration associates one semantic label of the semantic sublanguage with each symbol, providing another convenience mentioned later in discussing actions.

INT identifiers and class names must be declared before they are used.

Examples:

INT PRIMARY FACTOR TERM EXPRESSION

```
CLASS UNARYOPERATOR + -
CLASSLAB OPERAND I ROUTINEI N ROUTINEN PRIMARY ROUTINEP
```

13.8. Productions

Syntax

```
<production>      ::= <PL label> : <production>
                    | <left part> [ <right part> ]
                    <<action> list>
<left part>       ::=<<symb> list>
<right part>      ::= ">" [ <<symb> list> ]
<action>          ::= "see Section 13.9"

<symb>            ::= <source symbol> | <meta symbol>
                    | <int identifier> | <PL int>
                    | <class name>
```

A <left part> may contain at most 5 <symb>s in the list.
A <right part> may contain at most 3 <symb>s in the list.

Semantics: The first production to be executed is the first one. Productions are executed in order of occurrence except when this is changed by an action. A sequence of productions may act as a subroutine. See Section 13.9, actions 1 and 8.

A production is executed as follows:

1. The <symb>s in the left part are compared with the syntax components (second component) of the top records of the stack. A match occurs if one of the following holds for each <symb> in the left part:

   A) the <symb> is a <symbol> and the same <symbol> appears on the syntax component of the corresponding record.

   B) the <symb> is ANY (it matches any symbol on the stack).

   C) the <symb> is a class name and the syntax component of the corresponding stack record is a symbol in the class <class name> (cf Section 13.7).

   If a match occurs go on to step 2; if no match occurs, execution is finished.

2. If the right part occurs in the production, then the records matched in (1) are deleted from the stack. Any <symb>s appearing in the right part are then stacked, in left-to-right order, as follows:

   A) if the <symb> is ANY, I, S, N or a class name, it must also have appeared in the left part. The complete stack record, whose position corresponded to the rightmost

occurrence of the <symb> in the left part, is stacked.

B) if the <symb> is a source symbol, PL int or int identifier, a record is added to the stack and its second component becomes that symbol.

3. The actions are executed.

13.9. Actions

We now present the possible actions which can occur in a production.

1. CALL <PL label> Execute the productions starting at the one labeled by the <PL label>, and continue until the action RETURN is executed. This is thus just a subroutine call. It may be recursive. Restriction: the action EXEC <class name> may not appear after a CALL action in a production.

2. ERROR <integer> Print " ERROR <integer>".

3. EXEC <label> Begin executing the semantic subprogram of the pass at the statement labeled <label>. When the semantic statement SYNTAX is executed, return to the action following this one. The <label> may not be in a procedure or iterative statement of the pass.

4. EXEC <class name> The class name, which must have been declared in a CLASSLAB declaration, must also appear in the left part of any production in which this action appears. Consider the symbol in the stack corresponding to the topmost occurrence of the class name in the left part of the production. The semantic subprogram is executed beginning at the semantic label associated with this symbol in the declaration of the class name. Upon execution of the semantic statement SYNTAX, control returns to the production subprogram at the point following this action. Please note the restriction in action 1.

   Example. Suppose we have the declaration

   CLASSLAB SIGN + SPLUS - SMINUS

   and that the stack contains

   E + E - E (top of stack)

   and finally that a match has just occurred using the production

   E SIGN E SIGN E EXEC SIGN .

   Then the semantic subprogram will be executed beginning at label SMINUS.

5. GO <PL label> The production labeled <PL label> becomes the

current production and matching begins. Any actions following the GO action will never be executed.

6. HALT <integer> Print the message "HALT <integer>" and stop the program.

7. IF SIGNAL GO <PL label> If SIGNAL is TRUE (not zero), execute the GO <PL label> action (cf Section 13.6.2).

8. RETURN Return to the point after the last CALL executed (cf action 1).

9. SCAN If this pass is not in parallel with others, build the next atom of the source program, put it in SCANSYM (cf Section 12.1), and push it onto the stack (cf $section 13.6.1).

10. SCAN <integer> This is equivalent to SCAN SCAN ... SCAN <integer> times.

11. SCANNER <identifier> The identifier must name a scanner definition (cf Section 12.). Until another SCANNER action is executed, the source program will be scanned according to the scanner definition identified.

12. STAK <symbol> The symbol is pushed onto the stack (component 2 of the new record - cf Section 13.6.1).

13. STAK SCANSYM Push the symbol in SCANSYM onto the stack. (cf Section 12.1).

14. CODE GENERATION SYSTEM (CGS)

14.1. CODEAREAS

14.1.1 introduction

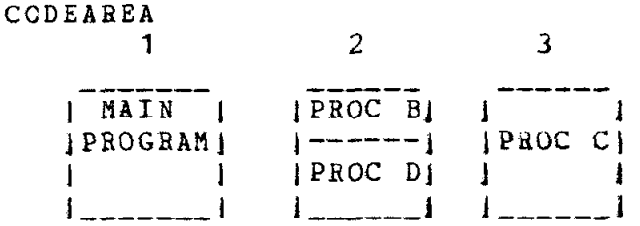
A CODEAREA is a table for storing code (machine language) as it is being generated at compile time. Code gets stored in a CODEAREA automatically as code bracket statements (cf Section 14.6) and expressions (cf Section 14.5) are executed. The compiler writer may also enter his own information into a CODEAREA with an ENTER statement (cf Section 14.1.5). At runtime, the contents of the CODEAREA becomes the program being run.

Any number of CODEAREAs may be used at compile time. They may contain code, tables of constants, or a mixture of both. Each CODEAREA becomes a named section, or CSECT, of the generated object module.

We make the following restriction on the use of CODEAREAs: the bytes of code for a subroutine should be contiguous. By a subroutine we mean a section or a program which may be "called" from many places, and which returns to the calling point when finished. To illustrate this, suppose a one-pass ALGOL compiler is compiling a program with the following structure:

```
BEGIN PROCEDURE B;
      BEGIN PROCEDURE C;
            BEGIN ... END;
       :
       :
      END;
      PROCEDURE D;
      BEGIN ... END;
 :
 :
END
```

Code for the main program and for procedures B and C must be generated into different CODEAREAS, while the code for procedure D may not be in the same area as the main program code. One possible configuration would be:



The main reason for the above restriction is to keep the code for

each logical part of the source program in contiguous bytes. This facilitates base register allocation and branching, which on the IBM 360 are complicated tasks.

The important points to remember about CODEAREAS are:

1. A CODEAREA at compile time is read-only storage at runtime.

2. The information is to be filled into the CODEAREA at compile time.

3. Each CODEAREA is a separate physical entity (a named section in OS 360 terminology).

4. At compile time, there is always one current CODEAREA into which code is being generated.

5. All CODEAREAS are in core during runtime (cf Section 14.9 for multiple coreloads).

The offset of a byte in a CODEAREA is the address of that byte in the CODEAREA. The first byte has offset 0, the second has offset 1, etc. Within CGS the address of any byte in a codearea is given by the pair (CODEAREA number, offset). CGS takes care of addressability problems when generating code.

14.1.2 register descriptions

CGS maintains a set of register descriptions for each CODEAREA. These register descriptions describe (at compile time) the runtime contents of the IBM 360 registers after the currently last instruction in the CODEAREA has been executed (at runtime). For example, suppose the statement

```
CODE(&GREG(1) = D)
```

has just been executed. This statement means "generate code to put the value of the runtime variable described by the DESCRIPTOR D into general register 1." The code for this is generated and put into the current CODEAREA. Then the register 1 description is changed to indicate that this value is now in register 1.

Execution of the above statement might also cause other descriptions to change. For example, if the runtime variable is not directly addressable, code must first be generated to load a register with the correct address (this is done by CGS automatically). When this happens, the description of that register is also changed.

A compiler writer may change and/or test register descriptions himself. All operations on them are explained in Section 14.4.

14.1.3 system variables connected with CODEAREAS

variable	type	meaning
&CODENO	BYTE	contains the number identifying the current CODEAREA.
&CODELOC	BYTE3	contains the offset of the next free byte in the current CODEAREA, and thus the number of bytes in the CODEAREA so far.

14.1.4 creating and switching CODEAREAS.

Evaluation of the <specfunc>

```
| &CREATECODEAREA |
```

causes a new CODEAREA to be created. The register descriptions of this new CODEAREA all initially indicate that the registers are empty. The value of the function designator is a BYTE value - the number assigned to the new CODEAREA. This number identifies the CODEAREA and is used to communicate with CGS.

The <specfunc>

```
| &USECODEAREA ( <expression> ) |
```

is evaluated as follows: the <expression> is evaluated, assigned to an internal BYTE variable I (say), and CODEAREA I (which must have already been created) becomes the current CODEAREA. This means that any code generated before the next USECODEAREA function designator executed, will be added to this CODEAREA. The value of the function designator is the BYTE value assigned to the previous current CODEAREA.

14.1.5 entering data into a CODEAREA

Code is entered into the current CODEAREA as code-bracketed statements are executed and code is produced. In addition, <specproc>s of the following form can be used:

```
| ENTER ( CODEAREA, [ <expression¹>, ] <expression²> ) |
```

This statement is executed as follows:

1. If <expression¹> is missing, then <expression²> is evaluated and added to the current CODEAREA at the next free byte with the proper alignment (cf Section 11 for alignment factors for different basic types). Variable &CODELOC is changed to the offset of the first free byte after the added bytes.

2. If <expression¹> is present, it is evaluated and assigned to an internal BYTE3 variable I (say). Next <expression²> is evaluated and the result is put in the CODEAREA at the offset I.

If the ENTER instruction is used and the entered data is actually code, it is the compiler writer's responsibility for updating the register descriptions.

Example. ENTER(CODEAREA, B)

14.1.6 initial conditions

Initially, CODEAREA 1 is the current CODEAREA and is the only one in existence. It may already contain some information; CODELOC may not initially be zero.

14.2 DATAAREAS

14.2.1 introduction

A DATAAREA is a runtime table for storing data - values corresponding to source language variables, temporary results, etc. In contrast to a CODEAREA which at runtime is read-only storage, a DATAAREA is read-write storage. Under certain circumstances, a DATAAREA can be initialized at compile time.

Storage is allocated in a DATAAREA to runtime variables through the allocate statements (cf Section 14.2.4). The allocated storage can be initialized at compile time by the &INIT or ENTER statements (cf Section 14.2.4).

The offset of a byte in a DATAAREA is the address of that byte within the DATAAREA. The first byte has offset 0, the second has offset 1, etc. Within CGS the address of any byte in a DATAAREA is given by the pair (DATAAREA number, offset within DATAAREA).

Actually, the BYTE numbers which identify DATAAREAS are different from those identifying CODEAREAS. Therefore a pair

(area number, offset)

uniquely addresses a byte or an AREA (CODEAREA or DATAAREA).

14.2.2 system variables connected with DATAAREAS

variable	type	meaning
&DATANO	BYTE	contains the number identifying the current DATAAREA.
&DATALOC	BYTE3	contains the offset of the next free byte in the current DATAAREA, and thus the number of bytes in the DATAAREA so far.

14.2.3 creating and switching DATAAREAS

The <specfunc>

```
┌-----------------------------------┐
| &CREATEDATAAREA [ ( DYNAMIC ) ] |
└-----------------------------------┘
```

creates a new, empty DATAAREA. The value of the function designator is a BYTE value which identifies the DATAAREA and which is used to communicate with CGS about the DATAAREA.

There are two types of DATAAREAS - STATIC and DYNAMIC.

1. If (DYNAMIC) is missing in the above function designator, the DATAAREA is STATIC. This means that it is a named section (control section) of the object module being generated; it exists throughout runtime (cf Section 14.9 for multiple coreloads.) it may be initialized at compile time. CGS handles all problems of addressing STATIC DATAAREAS.

2. If (DYNAMIC) is present, the DATAAREA is DYNAMIC. No named section for it exists in the object module being created and it cannot be initialized. Its function is to describe the format of a section of storage which may or may not exist at different stages of runtime. It thus is like a "DSECT" in an OS 360 assembly language program.

   One use of a DYNAMIC DATAAREA is for the variables and temporary locations associated with a procedure. At compile time storage can be allocated within the DATAAREA and code generated which uses the DATAAREA (even though no storage actually exists). At runtime, when the procedure is called, the necessary storage corresponding to the DATAAREA must be taken from free storage and used. Just before the procedure returns to the calling point, the storage is released again.

   Since DYNAMIC DATAAREAS are not always in core and may also appear in different locations, CGS needs some help in addressing variables in them. Briefly, the compiler writer must indicate a variable or register which contains the address of the DATAAREA. See Section 14.2.6 for full details.

The <specfunc>

```
| &USEDATAAREA ( <expression> ) |
```

is evaluated as follows: the <expression> is evaluated, assigned to an internal BYTE variable I (say), and DATAAREA I (which must have already been created) becomes the current DATAAREA. This means that any storage allocated or entered by an allocate or ENTER statement (cf Section 14.2.4) is entered into this DATAAREA until the next USEDATAAREA function designator is executed. Also, all storage needed for temporary results by CGS is allocated in the current DATAAREA. The value of the USEDATAAREA function designator is the BYTE number of the previous current DATAAREA.

14.2.4 allocating and initializing DATAAREA storage

Before reading this section glance over Section 14.3.

14.2.4.1 The <specproc> &ALLOCP allocates storage to one or more runtime variables of the same type.

Examples. To build a DESCRIPTOR for a halfword integer and allocate runtime storage for it, use

```
D = DESCRIPTOR(KIND=&HWI) ; &ALLOCP(D).
```

To build a DESCRIPTOR for a POINTER and allocate runtime storage in DATAAREA 3 for six POINTERS, use

```
D = DESCRIPTOR(KIND=&POINTER) ; &ALLOCP(D,6,DATAAREA 3) .
```

The syntax of the &ALLOCP <specproc> is

```
| &ALLOCP ( <DESCRIPTOR destination>          |
|             [ , <expression¹> ]             |
|           [ , DATAAREA <expression²> ] ) |
```

The default option for <expression¹> is 1. The default option for DATAAREA <expression²> is DATAAREA &DATANO (the current DATAAREA).

The statement accomplishes the following:

1. The DESCRIPTOR <destination> is checked. It must not describe a label, procedure or be undefined. The address of the variable must be completely undefined.

2. DATAAREA <expression²> becomes the current DATAAREA.

3. &DATALOC is increased, if necessary, to provide the proper alignment for the runtime variable described by the DESCRIPTOR <destination>.

4. The address (&DATANO,&DATALOC) becomes the basic address of the DESCRIPTOR <destination>.

5. <expression¹> is evaluated and assigned to an internal HWI variable I (say); the result must be nonnegative. &DATALOC is then increased to provide room for I runtime variables of the type specified by the DESCRIPTOR <destination> (If I = 0, nothing happens).

6. The DATAAREA which was current before this statement was executed becomes the current DATAAREA.

14.2.4.2 The &ALLOCF <specfunc> builds a DESCRIPTOR and then allocates runtime storage for it. The value of the function is the DESCRIPTOR.

Examples. To build and allocate storage for a halfword integer, use

```
D = &ALLOCF(&HWI) .
```

To build a DESCRIPTOR for a POINTER and allocate storage for 6 of them in DATAAREA 3, use

```
D = &ALLOCF(&POINTER,6,DATAAREA 3) .
```

To just align &DATALOC (current DATAAREA offset) on a doubleword boundary, use

```
&ALLOCF(&DWF,0) .
```

The syntax of the &ALLOCF <specfunc> is

```
┌──────────────────────────────────────────┐
│  &ALLOCF ( <expression⁰>                 │
│            [ , <expression¹> ]           │
│            [ ,  DATAAREA <expression²> ] )│
└──────────────────────────────────────────┘
```

It is evaluated as follows.

1. <expression⁰> is evaluated and assigned to an internal BYTE variable J (say). A new DESCRIPTOR D (say) is then generated with KIND = J.

2. The statement

```
&ALLOCP(D [ ,<expression¹>] [,DATAAREA <expression²>])
```

is then executed.

3. The value of the function is the DESCRIPTOR D. If its address is assigned to a POINTER variable, it is the programmers responsibility to release the storage for D when no longer needed. Otherwise the system takes care of it.

14.2.4.3 The &INIT <specproc> initializes runtime variables in a STATIC DATAAREA.

Examples. Let D be a DESCRIPTOR of a HWI value. To initialize the variable it describes with 0, use

&INIT(D,0) .

To initialize it and three following halfword integers with the current value of a compile time variable I, use

&INIT(D,4,I) .

Let PD be a DESCRIPTOR of a POINTER. To initialize the variable to point to itself, use

&INIT(PD, &ADD(PD)) (cf Section 14.3.4.5).

To initialize it to contain the address of CODEAREA 1, offset 4, use

&INIT(PD, &DDRESS(1,4)) (cf Section 14.3.1.1).

The syntax of the &INIT <specproc> is

```
┌-------------------------------------┐
|  &INIT ( <DESCRIPTOR destination>   |
|            [ , <expression¹> ]     |
|            , <expression²> )        |
|-------------------------------------|
|  &INIT ( <DESCRIPTOR destination>   |
|            [ , <expression¹> ]     |
|            ,<&DDRESS exp> )         |
└-------------------------------------┘
```

The default option for <expression¹> is 1. The second form is used if the runtime variable has type POINTER; the value to which it is initialized is the value of <&DDRESS exp> - cf Section 14.3.1.1). The first form is used if the runtime variable is not a pointer.

The statement is executed as follows:

1. The address of the runtime variable defined by the DESCRIPTOR <destination> is evaluated (at compiletime). It must

yield an address of the form (area number, offset). (this means for example that no indirect addressing may be specified.)

2. <expression¹> is evaluated and assigned to an internal HWI variable I (say); the result must be nonnegative.

3. <expression²> (or <&DDRESS exp> in the second case) is evaluated and assigned to an internal variable J (say) whose type is the same as that given by component KIND of the DESCRIPTOR <destination>.

4. The value of J is stored in the DATAAREA at the offset specified by the result of step 1, and in the following I - 1 runtime variables of the same KIND.

14.2.4.4 The ENTER DATAAREA <specproc> can be used to enter data into STATIC DATAAREAS. Its syntax is:

```
| ENTER ( DATAAREA, [ <expression¹>, ] <expression²> ) |
```

It is executed exactly like the ENTER CODEAREA statement (cf Section 14.1.5), except that a DATAAREA (which must be STATIC) is used instead of a CODEAREA.

Example. ENTER(DATAAREA,C)

14.2.5 initial conditions

Initially, DATAAREA 2 is the current DATAAREA and is the only one in existence. It is STATIC and may already contain some information.

14.2.6 addressing DYNAMIC DATAAREAS

since DYNAMIC DATAAREAS are not always in core - and since several copies may exist at any one time - CGS needs help in addressing them. There are two kinds of statements dealing with this problem; the first kind tells CGS that a DYNAMIC DATAAREA has been created (at runtime) and gives its location, the second kind tells CGS that a DATAAREA is no longer available.

14.2.6.1 Addressing new DATAAREAS. The following three <specfunc>s give CGS the address of a DATAAREA that can be referenced in the current CODEAREA only.

```
&DYNADD ( <DESCR exp> )
&DYNADD ( <DESCR exp> , <&DDRESS exp> )
&DYNADD ( <register no> , <&DDRESS exp> )
```

In the first case, the DESCRIPTOR must describe a &POINTER constant; the value of the pointer must be the address (in (area number, offset) form) of the DATAAREA which can now be referenced. In the second and third cases, the <&DDRESS exp> gives the address of the DATAAREA, while the actual place where this value resides is either at the address specified by the <DESCRIPTOR exp> or in register <register no>.

In all three cases the value of the <specfunc> is a pointer to a DESCRIPTOR of a &POINTER constant whose value is the address given.

14.2.6.2 releasing the DATAAREA. The <specproc>

```
&RELDYNADD ( <POINTER expr>)
```

tells CGS that the &POINTER constant described by the DESCRIPTOR pointed at by the <&POINTER expr> can no longer be used to reference data while executing the current CODEAREA.

14.3. The DESCRIPTOR

DESCRIPTOR is a structured type which is declared implicitly by the system. A variable of type DESCRIPTOR describes a runtime variable or value in terms of the IBM 360 basic data types. CGS provides several functions which alter, test and use DESCRIPTORS; the compiler writer should use these rather than try to perform these operations himself.

We use the word DESCRIPTOR for the structured type and also for a quantity of that structured type. When writing programs, the identifier "&D" can be used in place of "DESCRIPTOR".

During the code generation process, CGS maintains pointers to DESCRIPTORS which are being used to generate code. For example, if a DESCRIPTOR of a label has been used to generate a branch but the address of that label is still undefined, CGS records this fact and fixes the branch address later. Also, if a value is in a register, the register description points to a DESCRIPTOR of that value. For

this reason CGS places the following restriction on the use of DESCRIPTORS:

> A DESCRIPTOR being used by CGS should not be changed or moved to another location.

In order to be safe, a compiler writer should work with pointers to DESCRIPTORS, instead of the DESCRIPTORS themselves.

14.3.1 structure of the DESCRIPTOR

This section discusses the format of DESCRIPTORS and three related structured types.

14.3.1.1 STRUCTURE &DDRESS (BYTE AREA, BYTE3 OFFSET);

&DDRESS defines the basic address (BA) of a runtime variable in terms of a CODE or DATAAREA number (AREA) and an offset of the variable in the AREA (OFFSET). This is not the whole story on addressing; the DESCRIPTOR also allows for subscripting and indirect addressing.

14.3.1.2 STRUCTURE DESCRIPTOR (

```
        BYTE KIND,
        BYTE ADDRCONT,
        BYTE CONTROLS,
        BYTE REG ALT BYTE BYTELENG,
        &DDRESS ADDR ALT POINTER (&CONST) PC
                  ALT POINTER (&SUBSCR) PS,
        BYTE4 THEIRS);
```

Component KIND describes the basic kind of the runtime variable or quantity. The list below gives system identifiers of constants, their hex value (which may change; use the identifiers only) and the type of variable they describe:

identifier	value	meaning - the variable is
&UNDEF	00	undefined
&BYTE	01	one (8 bit) byte
&BYTE2	02	two contiguous (8 bit) bytes
&BYTE3	03	three contiguous (8 bit) bytes
&BYTE4	04	four contiguous (8 bit) bytes
&HWI	05	HalfWord Integer
&FWI	06	FullWord Integer
&FWF	07	FullWord Floating point number
&DWF	08	DoubleWord Floating point number
&DEC	09	DECimal integer
&POINTER	0A	address of something or 0
&BYTES	0B	1 to 256 contiguous bytes(components BYTELENG, PS, CONTROLS help describe how many bytes)
&PROC	10	procedure

&LABEL 20 label

Note that if you delete the first letter "&" from most of the system identifiers above, a CIL basic type is left (example - &HWI becomes HWI). In these cases, all attributes (ie. Length, alignment properties) for the runtime variable are the same as those for a value of the basic type.

Component ADDRCONT gives more information about addressing the runtime variable. It indicates whether the basic address (BA) is undefined, whether it is given by component ADDR, or whether it is a register. Subscripting and indirect addressing are also indicated. See 14.3.1.5.

Component CONTROLS contains miscellaneous bits used for different purposes. The following table gives system identifiers for constants, their hex values, and the meaning when an identifier is "anded" with component CONTROLS.

System identifier	hex value	meaning when identifier is "anded" with CONTROLS
&BL	01	for DESCRIPTORS of &BYTES only. If 0, number of bytes minus 1 is given in component BYTELENG otherwise the number of bytes is described by what PS points to.
&NEG	02	if not 0, negative of runtime value is desired.
&NOSAV	04	if not 0, save DESCRIPTOR, if 0, can be released after one use in code generation.
&ORD	08	if not 0, a saved register description points to DESCRIPTOR,
&OURS	10	if not 0, CGS created DESCRIPTOR
&LZ	20	(only when KIND is BYTES(1,2 or 3) or &POINTER and the value is in a register). If not 0, leading bytes of the register are 0.

Component REG indicates whether the value is in a register or not (cf Section 14.4.1):

0 = not in a register
1 through F mean general register 1 through 15
10 denotes general register 0
11 denotes floating register 0
12 denotes floating register 2
13 denotes floating register 4
14 denotes floating register 6

Component BYTELENG is used only if the KIND is &BYTES. It can contain the number of bytes minus 1 (if constant and less than 257). See component CONTROLS .

Component ADDR usually defines the basic address of the runtime variable. In certain cases, however, the basic address is defined by component ADDR of the quantity pointed at by component PC or PS (see

also component CONTROL.

Component_PC is used if the value is a constant. It points at a quantity of structured type &SUBSCR which gives the constant itself and its address.

Component_PS has two uses.

1. If the runtime variable is subscripted (cf component ADDRCONT), PS points at a quantity of structured type &SUBSCR which contains the basic address and a pointer to the subscript DESCRIPTOR.

2. If the runtime variable is of type BYTES and if component control "anded" with &BL is not 0, then PS points at a quantity of structured type &SUBSCR which contains the basic address and a pointer to a DESCRIPTOR or the number of bytes minus 1. Such DESCRIPTORS may not indicate subscripting.

14.3.1.3 STRUCTURE &CONST (
BYTE4 VALUE ALT &DDRESS ADDRVAL,
&DDRESS ADDR);

A quantity of type &CONST is used to help describe constants. The constant is held in component VALUE or ADDRVAL (if the constant is a relocateable address). The address of the constant is contained in ADDR. If ADDR.AREA and ADDR.OFFSET are both zero, the address is undefined.

14.3.1.4 STRUCTURE &SUBSCR(
POINTER (DESCRIPTOR) SUBDCR, &DDRESS ADDR);

a quantity of type &SUBSCR is used to help describe runtime variables which are subscripted or of type &BYTES (see below). Component ADDR contains the base address of the variable. If subscripting, &SUBSCR points to a DESCRIPTOR of the subscript. If not subscripting and the runtime variable is of type &BYTES, &SUBSCR points to a DESCRIPTOR of the number of bytes minus 1.

14.3.1.5 address description and format of DESCRIPTORS. This section describes just how the effective address is to be obtained from the basic address. Component ADDRCONT plays the key role here.

In the tables below, BA specifies that the basic address is given by component ADDR, while R indicates that the basic address is the register given by the number in ADDR.AREA. X specifies a subscript - its value is given by the DESCRIPTOR pointed at by the pointer PS.SUBDCR. "*" indicates indirect addressing. The format

number refers to the format of the DESCRIPTOR when ADDRCONT has the given value. The possible formats are given after the tables.

VALUE OF ADDRCONT AND MEANING IF KIND IS NOT &BYTES

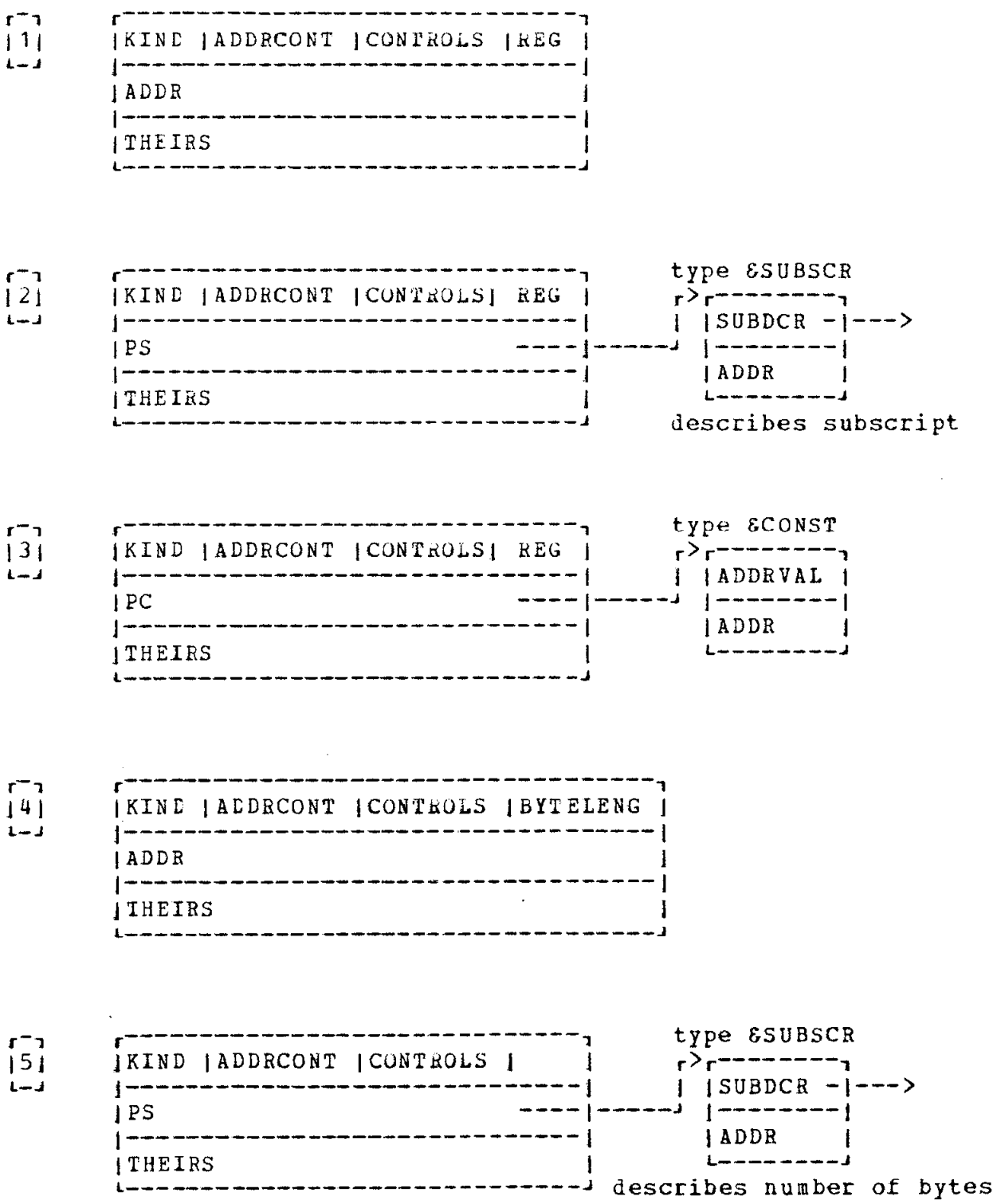
value	format	effective_address_is
0	1	_ (undefined)
1	3	BA (and the value is a constant)
2	1	BA
3	1	*BA
4	1	**BA
5	1	R
6	1	*R
7	1	**R
8	2	BA+X
9	2	*(BA+X)
A	2	(*BA)+X
B	2	*((*BA)+X)
C	2	(*R)+X
D	2	*((*R)+X)

VALUE OF ADDRCONT AND MEANING IF KIND IS &BYTES

value	format	effective_address_is
0	4	_ (undefined)
1	1	BA (value is a constant)
2	4 OR 5	BA
3	4 OR 5	*BA
4	4 OR 5	**BA
5	4 OR 5	R
6	4 OR 5	*R
7	4 OR 5	**R

Format 4 is used if the number of bytes minus 1 is contained in component BYTELENG ; otherwise format 5 is used.

Possible formats of a DESCRIPTOR



14.3.2 generating DESCRIPTORS

The DESCRIPTOR is a structured type, and a new quantity of that type can be generated and initialized in the usual manner. However it is easier and safer to initialize only component KIND and use the CGS operations to manipulate the rest. To aid in this, the system sets all components to 0 before initializing a new DESCRIPTOR, since zero is the natural initial state for its components. For example, if T is a table of DESCRIPTORS, then

```
T(2) = DESCRIPTOR(KIND=&LABEL)
```

puts in the second element a DESCRIPTOR of a label with an undefined address. If P is a POINTER variable, then

```
P = &D(KIND = &HWI)
```

allocates space for a new DESCRIPTOR of kind &HWI, sets all other components to zero, and puts the address of the DESCRIPTOR in P.

DESCRIPTORS may also be generated using the <specfunc> &ALLOCF (cf Section 14.2.4.2).

14.3.3 defining the basic address (BA)

Once component KIND is defined, there are several ways of filling in the basic address. Below, we assume that D is a DESCRIPTOR.

1. If the DESCRIPTOR defines a label or procedure, use it in code brackets (cf Sections 14.6.4 and 14.6.7). Example: CODE(D:) .

2. If the runtime variable is to be in a DATAAREA, use the <specproc> &ALLOCP or the <specfunc> &ALLOCF. Example: &ALLOCP(D).

3. If the runtime variable is external to the program being compiled, use the &EXTERN <specproc> (cf Section 14.3.6). Example: &EXTERN(D).

4. If the address to be used is already known, use the <specproc> &ASSIGNAD (cf Section 14.3.4.3). Example: &ASSIGNAD(D,&DDRESS(1,0)) (address of CODEAREA 1).

14.3.4 defining the effective address (EA)

Besides the basic address, the DESCRIPTOR can indicate indirect addressing and subscripting. The final address is called the effective address (EA). This section describes ways of indicating

effective addresses.

It is important to realize that the operations described here may generate code. For example, if an operation asks for subscripting for a DESCRIPTOR of a &BYTES variable, code must be generated to calculate the effective address because DESCRIPTORS of &BYTES variables do not allow subscripting. In general, CGS tries to postpone code generation as much as possible, since this usually produces better code.

Section 14.3.1.5 indicates, for each type of runtime variable, what kind of addressing the DESCRIPTOR can describe.

14.3.4.1 specifying subscripting. Syntax:

```
<DESCR exp>          ::= <DESCR exp¹> ( <DESCR exp²> )
                       | <DESCR exp¹> ( <expression> )
```

Semantics: A new DESCRIPTOR is generated. All of its components except those which help define the effective address are identical to those of <DESCR exp¹>. If EA is the effective address of <DESCR exp¹>, then the effective address of the new DESCRIPTOR is found as follows:

Case 1: <DESCR exp²> is present. The effective address is

EA + (runtime value described by <DESCR exp²>)

Case 2: <expr> is present. <expr> is evaluated and assigned to an internal FWI variable I (say). Then the effective address is

EA + I .

This may cause code to be generated. This depends on whether or not the new effective address can be described in a DESCRIPTOR. If <DESCR exp¹> is a CGS DESCRIPTOR, it will be released if possible (cf Section 14.8).
Examples. D1(D2) . D1(1) . D1(2)(6*I) is equivalent to D1(2+6*I).

14.3.4.2 specifying indirect addressing. The following <specfunc> is used to specify indirect addressing.

```
| &INDIR ( <DESCR exp> [ , <expression> ] ) |
```

The value of this function designator is a structured value of type DESCRIPTOR. All components, except those which have to do with addressing, are the same as those of <DESCR exp>. If EA is the effective address of <descr exp>, the effective address of the new DESCRIPTOR is

CONTENT(EA) .

If <expression> is present, it is assigned to component KIND of the new DESCRIPTOR.

This may cause code to be generated. This depends on whether or not the new effective address can be described in a DESCRIPTOR.

If <descr exp> is a CGS DESCRIPTOR, it will be released if possible (cf Section 14.8).

Examples. &INDIR(D).
&INDIR(D)(5) (indirect addressing followed by subscripting).
&INDIR(D(5)) (subscripting followed by indirect addressing).

14.3.4.3 using an existing address. The <specproc>

```
| &ASSIGNAD ( <destination> , <DESCR exp> ) |
```

puts the effective address of the DESCRIPTOR <DESCR exp> into the DESCRIPTOR <destination>. Only the address-describing components of <destination> are changed. Examples: &ASSIGNAD(D1,D2) . &ASSIGNAD(D1, &INDIR(D2)(1)) .

14.3.4.4 forcing code to be generated. The functions described in Sections 14.3.4.1 - 14.3.4.3 may cause code to be generated. The following <specfunc> indicates that code must be generated (if possible) to calculate the effective address.

```
| &EACALC ( <DESCR exp> ) |
```

The resulting value is a DESCRIPTOR which has all the characteristics of <DESCR exp> except that the EA specifies no subscripting and at most one level of indirect addressing (the address is in a register or in memory).

14.3.4.5 using an effective address as a value. Execution of the <specfunc> .

```
| &EAVAL ( <DESCR expr> ) |
```

yields a DESCRIPTOR with KIND &POINTER. The value it describes is the effective address of the <DESCR exp>. This may cause code to be generated.

The <specfunc>

```
| &ADD ( <DESCR exp> ) |
```

yields an &DDRESS value which is tne address contained in the DESCRIPTOR <DESCR exp>.

14.3.5 the length of &BYTES variables

The <specfunc> &LENGTH is used to indicate the number of bytes (minus 1) in a &BYTES runtime variable. Its syntax is

```
| &LENGTH ( <DESCR exp1>, <DESCR exp2> )   |
|------------------------------------------|
| &LENGTH ( <DESCR exp1>, <expression> )   |
```

It produces a DESCRIPTOR with KIND = &BYTES. The number of bytes minus 1 is given by the runtime variable described by <DESCR exp²> or by the current value of <expression>. All other components are the same as those of <DESCR exp¹>.

Examples: &LENGTH(D1,5) .
&length(&indir(d1), d2) .

14.3.6 runtime entry points and external references

When an OS 360 object module is being generated, one can specify entry points - bytes within this object module which may be referenced by other object modules - and external references - references to names which are not in this object module but which will be resolved by the OS linkage editor just before runtime.

14.3.6.1 The &ENTRY <specproc> is used to indicate an entry point. Its syntax is:

```
| &ENTRY ( <DESCR destination> , <STRING expr> ) |
```

It is executed as follows: The DESCRIPTOR destination must have an effective address of the form (AREA number, offset). The STRING expression is evaluated and assigned to an internal variable S (say) of type STRING(8). The value of S then becomes the name of the entry point.

Example: ENTRY(D1,'SIN') .

14.3.6.2 The &EXTERN <specproc> is used to indicate an external reference. The syntax is:

```
| &EXTERN ( <DESCR destination> , <STRING expr> ) |
```

It is executed as follows: The address in the DESCRIPTOR destination must be undefined. Space is allocated for a POINTER variable in the current DATAAREA, if STATIC, or DATAAREA 2 if DYNAMIC. At runtime this POINTER will contain the address of the external reference, the address of this POINTER becomes the BA of the DESCRIPTOR and indirect addressing is also indicated. The STRING expression is evaluated and assigned to a variable S (say) with type STRING(8). The value of S is then the name of the external address.

14.3.7 generating DESCRIPTORS for constants

CGS keeps a table of DESCRIPTORS for constants. All constants are stored in DATAAREA 2 - and only if they are actually needed at runtime. The following <specrunc>s all yield a value which is a POINTER to a DESCRIPTOR for a constant:

```
|       &CON   ( [<expr⁰>,] <expr¹> )          |
| &CON  ( [<expr⁰>,] <expr¹>,<expr²>,<expr³> ) |
|       &CON   ( <&DDRESS exp> )               |
```

The default option for <expr⁰> in the first two cases is &UNDEF. In these two cases, <epxr⁰> is evaluated and assigned to an internal BYTE variable I (say). The value of I then becomes the KIND of the DESCRIPTOR being created. The constant itself is then evaluated. In the first case it is <expr¹>; in the second case, <expr¹> is the integer part, <expr²> the fraction, and <expr³> the exponent. (all three must be integer-valued and the signs of <expr¹> and <expr²> must be the same). The constant is then converted to the KIND of the new DESCRIPTOR and inserted in it (if KIND = &UNDEF, the KIND is changed to the KIND of the constant.)

In the third case, a POINTER to a DESCRIPTOR of a &POINTER constant is generated; the value of the constant is the value of the <&DDRESS exp>.

Examples: to create a DESCRIPTOR of the constant 1.23x10-6 use

&CON(1.23*.000001) or &CON(1,23,-6).

To create a doubleword constant for it, use

```
&CON(&DWF,1,23,-6).
```

To create a constant whose value is the address of the next free byte in the current CODEAREA, use

```
&CON(&DDRESS(&CODENO, &CODELOC) ) .
```

14.4. Runtime registers and their descriptions

CGS maintains descriptions of the contents of the runtime registers as code is being generated. The description of a register consists mainly of a pointer to the DESCRIPTOR of the value in the register and some status bits which indicate how the register is being used.

For example, if the statement P = CODE(D+5) is executed, code is generated to add 5 to the value described by the DESCRIPTOR D, a new DESCRIPTOR D1 (say) is generated to describe the resulting value, and the address of D1 is stored in P. Suppose the resulting runtime value is in in general register 5. Then the description for register 5 will be changed to point to D1.

The compiler writer can leave most of the register handling to CGS, or he can make full use of the facilities described in this section to do his own register allocation.

14.4.1 register numbers and names.

Syntax:
<register no> ::= <BYTE expression>
<register name> ::= &GREG | &FREG | &REG(<expression>)

Semantics: The registers are numbered as follows:

```
1 - general register 1
2 - general register 2
: - : : : : : : : : :
: - : : : : : : : : :
F - general register 15
10- general register 0
11- floating register 0
12- floating register 2
13- floating register 4
14- floating register 6
```

In certain contexts, the system names &GREG and &FREG denote a general register and a floating register, respectively. The precise register to use is picked by CGS. Also, the construct &REG(I), where I is a BYTE expression, is used to denote register I in certain contexts.

14.4.2 general runtime register usage

CGS uses the usual OS 360 subroutine linkage conventions. A compiler writer need not follow them, but it is better if conventions are followed. When not actually linking, these linkage registers can be used for other purposes. The table below gives a brief explanation; a more complete description may be found in the IBM System/360 Operating System - Supervisor and Data Management Services (Form C28-6646), pages 9 - 16.

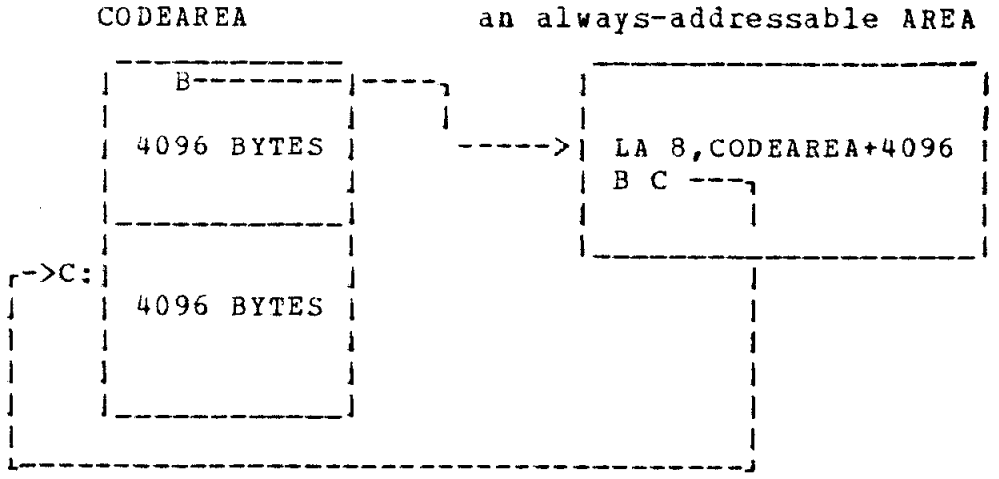
In addition, CGS requires two to three additional registers to be used as base registers at runtime. These contain the address of DATAAREA 2, the address of the current DATAAREA (if not 2 and if register 13 does not hold it), and the address of the current subroutine (or main program).

register	use
0	temporary or linkage: parameter. Not restored.
1	temporary or linkage: parameter or address of a parameter list. Not restored.
2-7	temporary. Restored.
8	temporary or used to provide addressability for instructions (see below). Restored.
9	address of a subprogram being executed (usually) the address of a CODEAREA). Restored.
10	temporary. Restored.
11	temporary, if current DATAAREA is 2 or its address is in register 13; otherwise address of current DATAAREA. Restored.
12	address of DATAAREA 2. Restored.
13	linkage: address or a SAVE AREA. This may also be the address of a DATAAREA if the SAVE AREA is part of it. Restored.
14	temporary or linkage: return address. Restored,
15	temporary or linkage: entry point when calling a program. Not restored.

Floating registers are not restored.

Those registers marked temporary may be used for any purpose. Upon return from a subprogram, those registers marked restored (reg 2-14) contain the same values they contained just before the subprogram was called.

The problem of addressing more than 4096 bytes of instructions is solved as follows. Register 9 always contains the base address of the subprogram being executed. If the code being executed does not lie within 4096 bytes of this address, register 8 contains the base address of the subprogram plus the multiple of 4096 bytes which gives the executed instructions addressability. Each branch is a single instruction. If the instruction being branched to is not addressable, then an indirect branch will occur. For example, the diagram below shows a branch to label C;



It is best to use registers 0 and 1 on a short-term temporary basis, since these registers are used often for parameters to subprograms.

14.4.3 register descriptions

At any time during code generation there is a set of current register descriptions which describe the runtime state of the registers after the last instruction entered in the current CODEAREA has been executed. As new instructions are generated, these register descriptions are changed to reflect the change in the runtime machine. There may be several sets of register descriptions at any time; when talking about register descriptions in general, we mean the current register descriptions unless otherwise stated.

A register description consists essentially of a pointer to a DESCRIPTOR of the value in the register and some "status" bits. These status bits are explained in the following table.

status	meaning
0	The register is &EMPTY (nothing in it).
1	The register is &USED. This means that it was formerly &NEW (see below) and the value in the register was used at least once since being put in the register. A USED value may be discarded (not saved) if a register is needed.
2	&SAVE the value in the register until further notice. If the register is needed for something else, the value must be saved; if its DESCRIPTOR contains an address, this location will be used, otherwise CGS assigns it a temporary location.
3	The value is &NEW. Once it is used to generate code it will be switched to &USED. When CGS generates a new value and its DESCRIPTOR, the register containing the value is set to &NEW.
4	The register is being used as a &FAST location for a variable or just contains a value which is not to be disturbed until further notice. For example,

registers 12 and 13 are &FAST registers (cf Section 14.4.2).

14.4.4 testing register status

Five functions, each with a single BYTE parameter which is a register number, test the status of the register specified:

<specfunc>	value is FALSE unless status is
&ISEMPTY (<register no>)	&EMPTY
&ISUSED (<register no>)	&USED
&ISSAVE (<register no>)	&SAVE
&ISNEW (<register no>)	&NEW
&ISFAST (<register no>)	&FAST

14.4.5 generating code to dump registers

When CGS needs a new register to hold a runtime value, it looks at the current register descriptions and uses one with the lowest status. (This is complicated somewhat by the fact that at times an even-odd register pair is needed, but we won't go into that here). The following table indicates what happens to the value in the register chosen.

register chosen has status	disposition of the old value in the register
0 (&EMPTY)	
1 (&USED)	the old value is lost
2 (&SAVE)	if the DESCRIPTOR associated with the register value has an undefined address, assign it an address. Then if the value is not a constant, generate instructions to store the value.
3 (&USED)	same as for &SAVE.
4 (&FAST)	never dumped in this manner. A &FAST register can be used for a different purpose only if its status is changed.

When a register is dumped, the register description status is set to &EMPTY.

The compiler writer may explicitly ask that code be generated

to store a register. The <specproc>

```
| &DUMPREG ( <register no>) |
```

does this for the register specified. The statement is executed as given in the above table. Note that &FAST registers may not be dumped.

14.4.6 generating code to load and use registers

The register names &FREG, &GREG and &REG(<expression>) may appear on the lefthand side of an assignment statement within code brackets. For example,

```
CODE(&REG = D )
```

is valid. The purpose of this statement is to generate code to load a value into a register. The execution of this statement is explained in detail in Section 14.6.2.

A register name &REG(<register no>) may also appear in a runtime expression within code brackets, to indicate that the contents of that register is to be used. See Section 14.5.

14.4.7 altering register descriptions

It is sometimes necessary to alter a register description without generating code. For example, after generating code for a function call, it may be necessary to tell CGS that the value of the function is in register 1.

14.4.7.1 changing the status to &EMPTY. The <specfunc>

```
| &EMPTY ( <register no>) |
```

changes the description of the register specified to &EMPTY. The DESCRIPTOR of the value in the register is changed to reflect the fact that it is no longer there and is then released if possible. The value of the function is a POINTER to the DESCRIPTOR of the value (0 if destroyed or there was none.)

14.4.7.2 changing the status to other than &EMPTY. Execution of The <specproc>s

```
| &USED( <register no>) |
|-----------------------|
| &SAVE( <register no>) |
|-----------------------|
```

&NEW (<register no>)
&FAST(<register no>)

change the status of the register to the desired status. The previous status must not have been &EMPTY.

14.4.7.3 indicating that a value is in a register. Execution of the <specproc>s

&USED(<register no>, <DESCR exp>)
&SAVE(<register no>, <DESCR exp>)
&NEW (<register no>, <DESCR exp>)
&FAST(<register no>, <DESCR exp>)

performs the following. The statement &EMPTY(<register no>) is exectued, emptying register <register no>. The status of the register is then changed to the desired status (procedure name), with <DESCR exp> being the DESCRIPTOR of the value in the register.

Notice that absolutely no code is generated by any of the procedures or functions described in this Section 14.4.7. The only purpose is to change a register description.

14.4.8 saving and restoring register descriptions

It is often advantageous to save a set of register descriptions for later use. For example, fewer instructions may be generated for a conditional statement if one indicates that the contents of the registers are the same at the beginning of the THEN statement and the ELSE statement. The following <specproc>s are used to manipulate the set of register descriptions. In all cases, the parameters P and P1 are <destination>s of type POINTER.

1. &SAVEREGS(P). Storage is allocated for a set of register descriptions. The current register descriptions are copied into the allocated storage. The address of the allocated storage is put in P.

2. &USEREGS(P). The set of register descriptions pointed at by P are copied into the current register description area.

3. &RESTREGS(P). Same as &USEREGS, but in addition, the storage pointed at by P is released and P is set to zero.

4. &JOINREGS(P). The set of register descriptions pointed at by

P are joined with the current register descriptions - for each register, if both descriptions are the same, the description remains; if the two descriptions are different the current register description is set to &EMPTY. The storage pointed at by P is released and P is set to zero.

5. &JOINREGS(P,P1). Join the register descriptions pointed at by P to those of P1 (as in 4.). Release the storage pointed at by P and set P to zero. Note: this does not change the current register descriptions.

6. &EXCHREGS(P). The register descriptions pointed at by P become the current register descriptions, while P is changed to point to the previous current ones.

When the current register descriptions are changed, CGS always checks to make sure that all register values are consistent with normal usage (cf Section 14.4.2). For example, register 8 and 9 are continually updated by CGS if necessary.

14.5. Code expressions

14.5.1 syntax

```
<runprimary>      ::= <constant> | <DESCR exp>
                    | <run variable>
                    | &REG ( <register no> )
                    | ( <runexp> )
<runfactor>       ::= <runprimary>
                    | <runprimary> ** <runfactor>
                    | <unary op> <runfactor>
<runexp>          ::= <runfactor>
                    | <runexp> <mult op> <runexp>
                    | <runexp> <add op> <runexp>
                    | <runexp> <bit op> <runexp>
                    | <runexp> <relational op> <runexp>
                    | <runexp> AND <runexp>
                    | <runexp> OR <runexp>
```

14.5.2 semantics

14.5.2.1 runtime primaries. A runtime primary yields a DESCRIPTOR of a runtime value. There are several types of runtime primaries:

<constant>. The DESCRIPTOR is a DESCRIPTOR for the constant. This does not necessarily mean that the constant occupies a place in storage at runtime. It will only appear in the object program if actually necessary.

<DESCR exp>. These have been discussed in Sections 14.3 and

14.3.4.1.

<run variable>. A <run variable> is a variable declared to be valid at runtime. If a primary is both a <run variable> and a <DESCR exp>, its use as a <DESCR exp> takes precedence.

&REG(<register no>). The register specified contains the value; its KIND is the KIND of the DESCRIPTOR associated with the register. If no DESCRIPTOR is associated with it currently, the KIND is assumed to be &FWI.

14.5.2.2 the operators. The operators available to operate on runtime values are exactly the same as those available to operate on compile time values. The precedence of the operators (cf Section 8.2.2) and the conversion of operands (cf Section 8.2.3) are also the same. The only difference is that evaluation of a <runexp> causes code to be generated for it. This code, when executed at runtime, will perform the desired evaluation. After the code is generated, a DESCRIPTOR is built to describe the runtime result.

14.5.3.3 using code brackets around expressions.

Syntax:
<DESCR exp> ::= CODE (<runexp>)

Semantics: Execution of this expression causes code to be generated to evaluate the <runexp> (if necessary). The result is the DESCRIPTOR for the runtime result of the <runexp>.

14.6. Code statements

Execution of a code statement causes code to be generated for the runtime statements appearing between the code brackets "CODE (" and ")". In the nonterminals defined below, the term "runstate" stands for "runtime statement". In general, a statement within code brackets has the same meaning as a similar statement outside, except that it indicates a runtime statement.

Syntax:

```
<code statement>  ::= CODE ( [ <<runstate> ;list> ] )

<runstate>        ::= <open runstate> | <closed runstate>

<open runstate>   ::= <runlabel definition> <open runstate>
                    | <open cond runstate>

<closed runstate> ::= [ <runlabel definition> ]
                      [ <closed runstate> ]
                      | <compound runstate>
```

```
                    | <assignment runstate>
                    | <closed cond runstate>
                    | <procedure run call>
                    | <control runstate>
                    | <procedure control>
```

14.6.1 compound runtime statements

Syntax:

```
<compound runstate> ::= BEGIN <<runstate> ;list> END
```

Semantics: A compound runtime statement is used to group several runtime statements into a single unit, just as a compound statement is used (cf Section 9.1).

14.6.2 assignment runtime statements

Syntax:

```
<assignment runstate> ::= <DESCR exp> = <runexp>
                        | <run variable> = <runexp>
                        | <register name> = <runexp>
```

Semantics: code is generated to evaluate the <runexp> and a DESCRIPTOR for the result is built. Code is then generated to store the result, depending on which of the above forms are used:

1. <DESCR exp> = <runexp>. Code is generated to convert the <runexp> to the KIND of the <DESCR exp> and to store the result in the location described by it (the address must be defined).

2. <run variable> = <runexp>. Code is generated to convert and store the <runexp> in the <run variable>.

3. <register name> (&GREG or &FREG) = <runexp>. An empty register is found; if necessary one is dumped. Code is then generated to store the <runexp> in this register. Its status is changed to &NEW. Code may be generated to convert the <runexp> to floating point (integer) if necessary, depending on which register name is used.

4. <register name> (&REG(<register no>) = <runexp>. If the register status is &EMPTY, we proceed as in (3) above. If not, code is generated to convert the <runexp> to the KIND of the DESCRIPTOR associated with the register and to store the value in it. The register status is not changed.

14.6.3 conditional runtime statements

Syntax:

```
<open cond runstate> ::= IF <runexp> THEN <closed runstate>
                           ELSE <open runstate>
                           | IF <runexp> THEN <runstate>
```

<closed cond runstate>::= IF <runexp> THEN <closed runstate>
ELSE <closed runstate>

Semantics: Execution of a conditional runtime statement causes code to be generated for it. Execution of this code at runtime will perform the operations in the usual manner (cf Section 8.2).

Example: IF D1 <= D2 THEN D1 = D2 ELSE GOIF D1

14.6.4 runtime label definitions

Syntax:
<runlabel definition> ::= <DESCR exp> :
| <DESCR exp> (0):
| <DESCR exp> (<POINTER destination>) :

Semantics: The <DESCR exp> must yield a DESCRIPTOR with KIND = &LABEL and with a completely undefined address. It is given the address (&CODENO,&CODELOC) - that is, the address of the next free byte in the current CODEAREA. Any already-generated references to this label will be fixed up - the address will be inserted in the branch instruction. (cf Section 14.6.5). The current register descriptions are changed as follows.

1. If the form <DESCR exp> : is used, the current register descriptions are changed as follows.

&USED registers are set to &EMPTY.

&SAVE and &FAST registers remain unchanged. It is up to the compiler writer to make sure that these registers are correctly loaded at all branches to this label. CGS takes care of registers 9 and 8.

If a register is &NEW an error message is printed. This is because the value has not been used and it is probably a mistake. Translation continues.

2. If the form <DESCR exp> (<POINTER destination>) : is used, the POINTER must point at a set of register descriptions. These become the current register descriptions and the <destination> is set to 0. The previously current register descriptions are released.

3. If the form <DESCR exp> (0): is used, the register descriptions remain unchanged. It is the compiler writer's responsibility to make sure that the descriptions are correct.

14.6.5 runtime control statements

Syntax:
<control runstate> ::= <goto op> <DESCR exp>
| GOIF <runexp> TO <DESCR exp>

| GOIFNOT <runexp> TO <DESCR exp>

Semantics: Execution of a runtime control statement causes an unconditional or conditional branch to be generated. The <DESCR exp> indicates where to branch to. If it has KIND &LABEL, its address need not yet be defined - CGS will automatically fix up the address when it becomes defined (cf Section 14.6.6). The <DESCR exp> may have KIND &POINTER, in which case its value is the address to branch to. In any case the address being branched to must lie in the CODEAREA where the branch occurs.

With the conditional branches GOIF and GOIFNOT, at runtime the branch will occur if the value of the <runexp> is not zero (TRUE) or zero (FALSE), respectively.

See Section 14.4.2 for a discussion of the instructions actually generated. CGS recognizes and produces better code in case the <runexp> has the form <runrelation> (cf Section 14.5).

14.6.6 runtime procedure calls

Syntax:
<procedure run call>::= <DESCR exp>

Semantics: The <DESCR exp> must yield a DESCRIPTOR with KIND &PROC. Execution proceeds as follows:

1. Code is generated to dump registers 14 and 15 if necessary.

2. Code is generated to load register 15 with the address defined by the <DESCR exp> (see below), if necessary. A DESCRIPTOR for it is built and associated with register 15 and the register status is changed to &USED.

3. A BALR 14,15 or a BAL 14,i(15) instruction is generated (see below).

If the address in the <DESCR exp> is not yet defined, the BALR instruction will be generated. When it becomes defined, the effective address can only be the basic address itself (no indirect addressing or subscripting).

If the address is already defined, and has the form A+X, (*A)+X or (**A)+X (cf Section 14.3.1) where X is a constant, the address A (*A or **A) will be loaded into register 15 and the instruction

BAL 14,value of X(15)

will be generated. Otherwise code is BALR 14,15 is generated.

14.6.7 runtime procedure entries and exits

Syntax:

```
<procedure control> ::= <procedure entry>
                      | <procedure exit>

<procedure entry> ::= <DESCR exp> :
<procedure exit>  ::= RETURN
```

Semantics: A <procedure entry> defines the address of a procedure entry point. The <DESCR exp> KIND must be &PROC. &CODELOC is increased until it is a multiple of 8 (on a doubleword boundary). Then the address (&CODENO, &CODELOC) is assigned to the DESCRIPTOR. In addition, the register descriptions are set as follows:

```
registers 0-11 &EMPTY
register 12 &FAST - contains address of DATAAREA 2
register 14 &EMPTY
register 15 &FAST contains address of the entry point.
```

Before executing a <procedure entry>, the compiler writer must do the following.

1. If this is not a multiple entry point in a procedure, switch to a CODEAREA which at this point is not being used.

2. If this is a multiple entry point in a procedure, generate the correct branch around this entry point.

After executing a <procedure entry>, the compiler writer must do the following.

1. Generate instructions to store the registers in the old SAVEAREA and to get a new SAVEAREA.

2. Generate instructions to move register 15 to register 9.

3. Change the register descriptions to reflect the proper register contents (especially registers 0,1,9,13, and 15.)

4. Generate instructions to take care of the procedure parameters.

5. Indicate the new current DATAAREA, if applicable.

Execution of a <procedure exit> causes the following code to be generated (conventional OS subprogram return).

```
L 13,4(13)        restore save area address
L 14,12(13)       return address in register 14
LM 2,12,28(13)    reload registers 2-12
BR 14             return
```

If this is the last instruction to be generated in this procedure the compiler writer should switch to another CODEAREA and perhaps DATAAREA. This CODEAREA can now be used for another

procedure.

14.7. Temporary runtime storage

At times CGS must temporarily store values (for example, if a register must be dumped). When this occurs, CGS allocates storage in the current DATAAREA, with the aid of the &ALLOCP statement (cf Section 14.2.4). This storage remains in existence for this purpose as long as the DESCRIPTOR of the value does. When the DESCRIPTOR is released, CGS will use the storage assigned to it for other temporary values.

14.8. When CGS releases DESCRIPTORS

CGS is continually generating DESCRIPTORS. If these are allocated new space, bit &OURS is set to 1, as soon as such a DESCRIPTOR is used in the code generation process, it can be released. Should the compiler writer wish to save it, he should set bit &NOSAV to 1. It is then his responsibility to release it.

A more detailed explanation will appear in a later version.

14.9. Specifying multiple coreloads

This Section will be completed at a later date.

This appendix gives the types of permissable operands for the binary and unary operators. In the tables below, B1, B2, B3, B4 and BS stand for BYTE, BYTE2, BYTE3, BYTE4 and BYTES(I) (for some I), respectively. P stands for POINTER.

Each row represents a left-hand operand, each column a right-hand operand of the operator. The corresponding table element is either blank - which means that that particular left-right pair is not valid - or is some type. In the latter case, before the operation is performed the two operands are converted to this type (as explained in Section 8.2.3). In addition, the result of the operation has that type.

+	B1	B2	B3	B4	BS	HWI	FWI	FWF	DWF	DEC	P
B1	HWI	FWI	FWI	FWI	FWI	HWI	FWI	FWF	DWF	DEC	P
B2	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	P
B3	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	P
B4	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	P
BS	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	P
HWI	HWI	FWI	FWI	FWI	FWI	HWI	FWI	FWF	DWF	DEC	P
FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	P
FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWI	FWF	DWF	DEC	
DWF	DWF	DWF	DWF	DWF	DWF	DWF	DWF	FWF	DWF	DEC	
DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	
P	P	P	P	P	P	P	P				

UNARY +	B1	B2	B3	B4	BS	HWI	FWI	FWF	DWF	DEC	P
	B1	B2	B3	B4	BS	HWI	FWI	FWF	DWF	DEC	P

-	B1	B2	B3	B4	BS	HWI	FWI	FWF	DWF	DEC	P
B1	HWI	FWI	FWI	FWI	FWI	HWI	FWI	FWF	DWF	DEC	
B2	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
B3	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
B4	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
BS	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
HWI	HWI	FWI	FWI	FWI	FWI	HWI	FWI	FWF	DWF	DEC	
FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWI	FWF	DWF	DEC	
DWF	DWF	DWF	DWF	DWF	DWF	DWF	DWF	FWF	DWF	DEC	
DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	
P	P	P	P	P	P	P					

UNARY -	B1	B2	B3	B4	BS	HWI	FWI	FWF	DWF	DEC	P
	HWI	FWI	FWI	FWI	FWI	HWI	FWI	FWF	DWF	DEC	

*	B1	B2	B3	B4	BS	HWI	FWI	FWF	DWF	DEC	P
B1	HWI	FWI	FWI	FWI	FWI	HWI	FWI	FWF	DWF	DEC	
B2	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
B3	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
B4	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
BS	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
HWI	HWI	FWI	FWI	FWI	FWI	HWI	FWI	FWF	DWF	DEC	
FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWI	FWF	DWF	DEC	
FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWI	FWF	DWF	DEC	
DWF	DWF	DWF	DWF	DWF	DWF	DWF	DWF	FWF	DWF	DEC	
DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	
P											

/	B1	B2	B3	B4	BS	HWI	FWI	FWF	DWF	DEC	P
B1	FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
B2	FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
B3	FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
B4	FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
BS	FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
HWI	FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
FWI	FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	FWF	DWF	DEC	
DWF	DWF	DWF	DWF	DWF	DWF	DWF	DWF	DWF	DWF	DEC	
DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	
P											

bits operators BITAND, BITOR, BITEXOR.

	B1	B2	B3	B4	BS	HWI	FWI	FWF	DWF	DEC	P
B1	B1	B2	B3	B4	BS	B2	B4	B4	BS	BS	
B2	B2	B2	B3	B4	BS	B2	B4	B4	BS	BS	
B3	B3	B3	B3	B4	BS	B3	B4	B4	BS	BS	
B4	B4	B4	B4	B4	B4	B3	B4	B4	BS	BS	
BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	
HWI	B2	B2	B3	B4	BS	B2	B4	B4	BS	BS	
FWI	B4	B4	B4	B4	BS	B4	B4	B4	BS	BS	
FWF	B4	B4	B4	B4	BS	B4	B4	B4	BS	BS	
DWF	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	
DEC	BS	BS	BS	BS	BS	BS	BS	BS	BS	BS	
P											

Exponentiation A**B. If A is HWI, FWI, or a bits type. And B is a positive integer constant, the result is FWI. Otherwise the result is DWF. A and B can have any type except POINTER and STRING.

REM and // are explained in section 8.2.4.

CAT	B2	STRING
B2	STRING	STRING
STRING	STRING	STRING

With the CAT operator, a BYTE2 operand is assumed to be an atom, and the string of characters it represents is used.

)	
GO	
ATOM	10.4
BEGINPASS	9.6
CALLPASS	9.6
CHARMODE	9.8
COMPLETE	9.6
DESCRIPTOR	14.3.1.2
FALSE	5.3
L0	7.1
L1	7.1
L2	7.1
L3	7.1
L4	7.1
R0	7.1
R1	7.1
R2	7.1
NORMODE	9.8
SCAN	9.8
SCANSYM	12.1
TRUE	5.3
&ADD	14.3.4.5
&ALLOCF	14.2.4.2
&ALLOCP	14.2.4.1
&ASSIGNAD	14.3.4.3
&BINT	9.9.3
&BL	14.3.1.2
&BYTE	14.3.1.2
&BYTE2	14.3.1.2
&BYTE3	14.3.1.2
&BYTE4	14.3.1.2
&BYTES	14.3.1.2
&CLINE	12.6
&CODELOC	14.1.3
&CODENO	14.1.3
&CON	14.3.7
&CONST	14.3.1.3
&CREATECODEAREA	14.1.4
&CREATEDATAAREA	14.2.3
&D	14.3
&DDRESS	14.3.1.1
&DATALOC	14.2.2
&DATANO	14.2.2
&DYNADD	14.2.6.1
&DEC	14.3.1.2
&DECT	9.9.3
&DUMPREG	14.4.5
&DWF	14.3.1.2
&EACALC	14.3.4.4
&EAVAL	14.3.4.5
&EMPTY	14.4.7.1
&ENTRY	14.3.6.1
&EXCHREGS	14.4.8
&EXTERN	14.3.6.2
&FAST	14.4.7.2
&FLPT	9.9.3
&FREG	14.4.1
&FWF	14.3.1.2
&FWI	14.3.1.2
&GREG	14.4.1
&HEXT	9.9.3
&HWI	14.3.1.2
&IN	9.9.1
&INDIR	14.3.4.2
&INIT	14.2.4.3
&INLINE	12.6
&INTDIC	10.4
&ISEMPTY	14.4.4
&ISFAST	14.4.4
&ISNEW	14.4.4
&ISSAVE	14.4.4
&ISUSED	14.4.4
&JOINREGS	14.4.8
&LABEL	14.3.1.2
&LENGTH	14.3.5
&LZ	14.3.1.2
&NEG	14.3.1.2
&NEW	14.4.7.2
&NLINE	12.6
&NOSAV	14.3.1.2
&OCTT	9.9.3
&ORD	14.3.1.2
&OURS	14.3.1.2
&OUT	9.9.2
&OUTDESCR	9.9.2
&POINTER	14.3.1.2
&PROC	14.3.1.2
&REG	14.4.1
&RELDYNADD	14.2.6.2
&RELEASE	9.10
&RESTREGS	14.4.8
&SAVEREGS	14.4.8
&SAVE	14.4.7.2
&SCLINE	12.6
&SUBSCR	14.3.1.4
&TBIN	9.9.3
&TDEC	9.9.3
&TEXT	9.9.3
&THEX	9.9.3
&TOCT	9.9.3
&TYPE	10.4
&UNDEF	14.3.1.2
&USECODEAREA	14.1.4
&USED	14.4.7.2
&USEDATAAREA	14.2.3
&USEREGS	14.4.8

The following identifiers are used to name components of

system structured types.

ADDR	14.3.1.2
ADDR	14.3.1.3
ADDR	14.3.1.4
ADDRCONT	14.3.1.2
ADDRVAL	14.3.1.3
AREA	14.3.1.1
BYTELENG	14.3.1.2
CONTROLS	14.3.1.2
KIND	14.3.1.2
OFFSET	14.3.1.1
PC	14.3.1.2
PS	14.3.1.2
REG	14.3.1.2
SUBDCR	14.3.1.4
THEIRS	14.3.1.2
VALUE	14.3.1.3

APPENDIX C. PROGRAM EXAMPLES

Example 1. This example illustrates basic declarations, assignment statements and iterative statements. It computes and prints factorial N,for N=1,...,10.

```
BEGIN FWI I,N;      /* I and N are Fullword Integers */
   I = 1;
   FOR N = 1 UNTIL 10 DO
       BEGIN  I = I*N;
              &OUT( 'FACTORIAL', N, I)
       END;
END;
```

Example 2. This example is a direct translation from ALGOL into CIL of Knuth's algorithm for calculating the day and month of Easter, given the year (cf Comm. ACM 5 (April 62), 209).

```
PROCEDURE EASTER( HWI YEAR,      /*input  */
                      MONTH,     /*output */
                      DAY);      /*output */
 BEGIN HWI GOLDENNUMBER, CENTURY, GREGORIANCORRECTION,
             CLAVIAN CORRECTION, EXTRADAYS, EPACT;
     GOLDENNUMBER = YEAR REM 19 + 1;
     IF YEAR > 1582
     THEN BEGIN CENTURY = YEAR // 100 + 1;
          GREGORIANCORRECTION =  (3 * CENTURY) // 4 - 12;
          CLAVIANCORRECTION = (CENTURY-16- (CENTURY-18)//25) // 3;
          EXTRADAYS = (5*YEAR) // 4 - GREGORIANCORRECTION - 10;
          EPACT = (11*GOLDENNUMBER + 20 + CLAVIANCORRECTION
                   - GREGORIAN CORRECTION) REM 30;
          IF EPACT <= 0 THEN EPACT = EPACT + 30;
          IF (EPACT = 25 AND GOLDENNUMBER > 11) OR EPACT = 24
          THEN EPACT = EPACT + 1;
          END
     ELSE BEGIN EXTRADAYS = (5*YEAR) / 4;
                EPACT = (11*GOLDENNUMBER - 4) REM 30 + 1;
          END;
     DAY = 4  - EPACT;
     IF DAY < 21 THEN DAY = DAY + 30;
     DAY = DAY + 7 - (EXTRADAYS+DAY) REM 7;
     IF DAY > 31 THEN BEGIN MONTH = 4; DAY = DAY - 31 END
END;
```

Example 3. This example illustrates one use of tables, BYTES variables and SUBBYTE designators. In JACM January 1962, Stephan Warshall gave the following algorithm for computing M* = M*M*...*M if M is a n by n Boolean matrix:

1. Set i = 1.

2. For all j such that M(j,i) = 1
set M(j,k) = M(j,k) OR M(i,k) for all k.

3. Increment i by 1.

4. If i <= n, go to step 2; otherwise stop.

We give to ways of implementing this in CIL.

```
PROCEDURE MSTAR( BYTES TABLE M; FWI N);
   /* M is a table of records, each of type BYTES(N) (a
      sequence of N 8-bit bytes). N is between 1 and 256.
      For I,J = 1,...,N, SUBBYTE(M(J),I-1,1)
      is the matrix element M(J,I) and will take on
      only the values 0 or 1. */

 BEGIN
   FWI I,J;                    /*I,J are FullWord Integers.*/
   FOR I = 0 UNTIL N-1 DO      /*loop on I */
       FOR J = 1 UNTIL N DO /*loop on J */
           IF SUBBYTE(M(J),I,1) = 1
           THEN M(J) = M(J) BITOR M(I+1);
 END
```

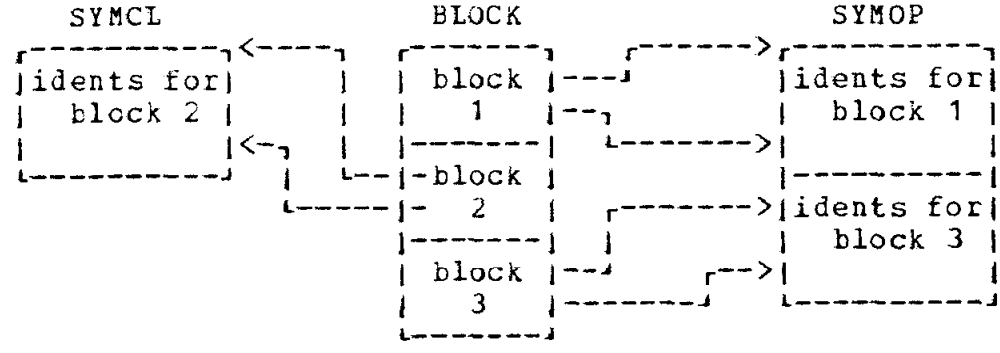
```
PROCEDURE MSTAR1( BYTES TABLE M; FWI N);
   /* this is as in the above case. However this time each of the
      8 bits in a byte of a record M(I) represents a matrix
      element. Thus the matrix represented can be 256*8 by 256*8.
      For I,J = 1,...,N, if K = (J-1) REM 8 + 1
      then bit K of the byte SUBBYTE(M(I), (J-1) // 8,1)
      represents the matrix element M(I,J). */

 BEGIN
   FWI I,J,K,L;
   BYTES(8) MASK;                  /* MASK is a sequence of 8 bytes */
  MASK = X'8040201008040201'; /* which is used to isolate a
                                 single bit of an 8-bit byte. Thus
                                 SUBBYTE(MASK,K,1) BITAND B
                                 yields the value (0 or not zero)
                                 of the K+1th bit of the BYTE
                                 variable B for K=0,...,7. */

   FOR I = 1 UNTIL N DO
```

```
    BEGIN K = (I-1) REM 8;
          L = (I-1) // 8;
          FOR J = 1 UNTIL N DO
              IF SUBBYTE(M(J),L,1) BITAND SUBBYTE(MASK,K,1)
              THEN M(J) = M(J) BITOR M(I);
    END
END
```

Example 4. This example illustrates the use of tables, structures and pointer variables. We wish to describe the symbol tables necessary to implement ALGOL block structure. Blocks will be numbered, starting with 1, in the order of their BEGINS. When a block is open, its identifiers will be stored in table SYMOP. When a block is closed, the records for identifiers in it will be moved from SYMOP to table SYMCL. All records for a block are contiguous. A table BLOCK helps to indicate where the records for each block are. For example, if we have so far parsed

```
BEGIN COMMENT block 1;
        :
        BEGIN COMMENT block 2;
                :
        END;
        BEGIN COMMENT block 3;
                :
```

the tables will look like



The declarations necessary are:

```
STRUCTURE SYMSTR(      /*structure of SYMOLD, SYMNEW record*/
       BYTE2 AT,       /*atom for identifier*/
       BYTE TYPE,      /*type of identifier*/
       BYTE BLOCKNO); /*block number in which declared*/

DYNAMIC SYMSTR TABLE 50 SYMOP;  /*table for identifiers in open
                                  blocks*/

DYNAMIC SYMSTR TABLE 99 SYMCL;  /*table for ids in closed blocks*/

STRUCTURE BLKSTR(      /*structure of BLOCK table record.*/
       BYTE BLOCKNO,   /*block number*/
       BYTE BLOCKSU,   /*surrounding block number*/
       POINTER PF,     /*to first record for block*/
       POINTER PL,     /*to last record for block (0 if none)*/
       BYTE TAB);      /*0= block in SYMOP, 1 = SYMCL.*/

DYNAMIC BLKSTR 50 BLOCK; /*table to control block structure*/
```

```
BYTE BLKCUR, BLKLAST;     /*current block number and last block
                          number assigned. Both are initially 0*/

POINTER (BLKSTR) B;       /*pointer to records of type BLKSTR.*/
POINTER (SYMSTR) P1,P;    /*pointer to records of type SYMSTR*/

BYTE AT, TYPE;            /*global variables.*/
```

The following should perhaps be explained. If P is a pointer variable pointing to some structured type record, and if X is the name of some component of that structured type, then

P.X

is a reference to the component X of the record pointed at by P. In addition, we assume there is a stack operating in the usual manner. L0 and L1 refer to the top and second stack records before the last matching of the stack with a production began. R0 and R1 refer to the current top and second stack records.

Two semantic routines are used to open new blocks and close blocks when entirely parsed:

```
SOPEN: /*this routine is called when a new BEGIN for a block
         is scanned. It adds a new record for the new  block in
         table BLOCK and fixes current block number.  */
 BLKLAST= BLKLAST+1; /*fix up the last block number - */
 ENTER(BLOCK, BLKSTR(BLKCUR,R0.BLKNO,0,0,0);
                     /*add the record for the new block*/
 BLKCUR = BLKLAST;   /*fix up current block number. */
 SYNTAX;             /*return to productions*/


SCLOSE: /*this semantic routine is called when BEGIN END is
          on the stack. It moves the records for this block from
          table SYMOP to SYMCL and fixes everything up. */
 B = @ BLOCK(BLKCUR); /*save the address of BLOCK record
                      for current block in B.*/
 IF B.PF              /*if this pointer is non-zero, we have some
 THEN BEGIN           /*record to move to SYMCL. */
       P1 = TALLY(SYMCL,0,BACK); /*save address of current last*/
                                 /*record of SYMCL.*/
       FOR P IN SYMOP FROM B.PF TO B.PL DO /*move the necessary*/
            ENTER(SYMCL,&C(P) );           /*records from
                                           SYMOP to SYMCL*/
       DELETE(SYMOP,B.PF);       /*delete the moved records*/
       B.PL = TALLY(SYMCL,P1);   /*now fix up the block record*/
       B.PF = TALLY(SYMCL,0,BACK); to point to the new records
                                   IN symcl.*/
     end;
 E.TAB = 1;           /*the records are now in SYMCL.*/
 BLKCUR = BLOCK(BLKCUR).BLOCKSU; /*new current block is the */
 SYNTAX;              /*previous surrounding one. */
```

Two procedures are used to enter records into the symbol tables and to look for records for identifiers:

```
PROCEDURE DEC; /*this procedure enters a record for identifier
                 AT with type TYPE for block number
                 BLKCUR.*/
 begin pointer p;
 P = ENTER(SYMOP, SYMSTR(AT,TYPE,BLKCUR)); /*enter the record,
                 put its address in P.*/
 IF P=0 THEN BLOCK(BLKCUR).PF=P; /*fix up the block structure*/
 BLOCK(BLKCUR).PL=P;             /*table record for this block.*/
 END;


PROCEDURE FIND; /*this routine looks in block BLKCUR and
                  surrounding blocks for an identifier named AT.
                  If found, P = address of its record; otherwise
                  P=0. BLKCUR, AT and P are global.*/
 BEGIN  BYTE K;
        POINTER(BLKSTR) B;
 P = 0; K = BLKCUR;   /*assume we can't find AT (P=0) and
                        initialize K to current block number*/
 WHILE K DO           /*we try current block and each
                        surrounding block, in succession*/
  BEGIN B = @ BLOCK(K); /*save address of block record*/
        IF B.TAB       /*we look for the identifier in the records
                         records for the block - in SYMCL if block
                         is closed, or SYMOP if open.*/
        THEN P = LOOK(SYMCL.AT, AT FROM B.PF TO B.PL)
        ELSE P = LOOK(SYMOP.AT, AT FROM B.PF TO B.PL);
        IF P                /*if P=0, AT wasn't in block, so*/
        THEN K = 0          /*set K to surrounding block number*/
        ELSE K = B.BLOCKSU /*otherwise we are done - set K to*/
  END;                      /*0 to end the WHILE statemnt*/
```

Example 5. This example illustrates the use of code brackets to generate code for conditional statements of the usual form. We assume that IF, THEN and ELSE are reserved words, that BE and S are INTS for Boolean expression and statement respectively, and that ENDIF is a class name for symbols which can end a conditional statement. The productions used here (we only list the ones necessary for illustration) are

```
IF BE THEN    > THEN       EXEC SBE      SCAN GO BEGINSTATEMENT

THEN S ELSE   > ELSE       EXEC STHELSE  SCAN GO BEGINSTATEMENT

THEN S ENDIF  > S ENDIF    EXEC SIFEND         GO ENDSTATEMENT

ELSE S ENDIF  > S ENDIF    EXEC SIFEND         GO ENDSTATEMENT
```

The following semantic routines generate code for conditional statements, without caring about the contents of the runtime registers. We assume the main stack has a component D which can be a pointer to a DESCRIPTOR.

```
SBE: /*stack contained IF BE THEN and L1.D contains a
  pointer to a DESCRIPTOR for BE. */
 R0.D =                          /*generate a new label to jump to*/
  DESCRIPTOR(KIND=&LABEL);       /*if BE is false and stack it.*/
 CODE(GOIFNOT L1.D TO R0.D);     /*generate a branch-on-BE-false*/
                                 /*to the label.*/
 SYNTAX;                         /*return to productions.*/

STHELSE: /*stack contained THEN S ELSE and we assume that the
      code for statement S has already been generated. */
 R0.D =                          /*generate a new label to jump to*/
  DESCRIPTOR(KIND=&LABEL);       /*after S is executed, stack it.*/
 CODE(GO R0.D);                  /*generate the branch to it.*/
 CODE(L2.D:);                    /*derine the address of the label*/
                                 /*to branch to if BE is false.*/
                                 /*CGS sets register descrptions*/
                                 /*to &EMPTY and fixes any*/
                                 /*previous branches to the label.*/
 RELEASE(L2.D);                  /*label is no longer needed-relese*/
 SYNTAX;                         /*it, return to productions.*/

SIFEND: /* stack contained THEN S ENDIF or ELSE S ENDIF
     and we assume code for statement S has been generated.
     L2.D contains a pointer to a DESCRIPTOR for an
     internal label for statement following ENDIF. */
 CODE(L2.D:);                    /*define the address of the label to
                                 branch to if BE is false (or after
                                 the THEN statement has been
                                 executed). Reg descriptions set to
                                 &EMPTY and previous branches to
                                 label are fixed up. */
 &RELEASE(L2.D);                 /*release the DESCRIPTOR.*/
```

```
  SYNTAX;                          /*return to productions.*/
```

The following semantic routines can be used in place of those above. They illustrate the use of the register descriptions to generate better code. In addition to component D, we assume that the main stack contains a pointer component which will point to register descriptions.

```
SBE: /*stack is as previous case*/
R0.D = DESCRIPTOR(KIND=&LABEL); /*as in previous case*/
CODE(GOIFNOT L1.D TO R0.D);  /*as in previous case*/
&SAVEREGS(R0.P);             /*save the current register descrip
                               tions for later use.*/
SYNTAX;

STHELSE: /*as in previous case, but L2.P contains a pointer
      to register descriptions as they were at the beginning of
      the THEN statement.*/
R0.D = DESCRIPTOR(KIND=&LABEL); /*as in previous case*/
CODE(GO R0.D);               /*as in previous case*/
&EXCHREGS(L2.P);             /*save the current register descrip
                               tions for later use and make the
                               current ones the same as they were
                               for the THEN statement.*/
R0.P=L2.P;                   /*make sure its stacked right.*/
CODE(L2.D(0):)               /*define label - but leave register
RELEASE(L2.D);               /*descriptions alone.*/
SYNTAX;

SIFEND: /*as in previous case, but l2.p contains pointer to descrip
          tions of registers as they were upon the branch-on-false
          or the branch after the THEN statement.*/
&JOINREGS(L2.P);             /*join the register descriptions
                               with current ones, since these
                               describe the only places that brnch
                               to here.*/
CODE(L2.D(0):);              /* as before, but leave register
&RELEASE(L2.D);              descriptions alone.*/
SYNTAX;
```