PL/I

TECHNICAL REPORT

TR 25.095
30 June 1969

ULD VERSION III

# FORMAL DEFINITION OF THE PL/I COMPILE TIME FACILITIES

M. FLECK

IBM LABORATORY VIENNA

N O T E

This document is not an official PL/I Language Specification. For information concerning the official interpretation the reader is referred to the PL/I Language Specifications, Form No. Y33-6003-1.

IBM LABORATORY VIENNA,
Austria

# FORMAL DEFINITION OF THE PL/I COMPILE TIME FACILITIES

by

M. FLECK

ABSTRACT

This report supplements the formal definition of PL/I by a formal definition of the PL/I compile time facilities. The concrete syntax, its abstract representation, and the abstract syntax of the PL/I compile time facilities are specified. A function is defined which maps a concrete PL/I compile time program into an abstract compile time program and an abstract machine is given which interprets abstract compile time programs.

Locator Terms for IBM Subject Index

PL/I
Compile Time
Formal Definition
Syntax, concrete
Syntax, abstract
21 PROGRAMMING

TR 25.095

30 June 1969

## PREFACE

This document is an updated version of:

/1/ FLECK, M., NEUHOLD, E.: Formal Definition of the PL/I Compile Time Facilities. IBM Laboratory Vienna, Techn. Report TR 25.080, 28 June 1968.

It is part of a series of documents (ULD Version III) presenting the formal definition of syntax and semantics of PL/I:

/2/ FLECK, M.: Formal Definition of the PL/I Compile Time Facilities (ULD Version III). IBM Laboratory Vienna, Techn. Report TR 25.095, 30 June 1969.

/3/ URSCHLER, G.: Concrete Syntax of PL/I (ULD Version III). IBM Laboratory Vienna, Techn. Report TR 25.096, 30 June 1969.

/4/ URSCHLER, G.: Translation of PL/I into Abstract Text (ULD Version III). IBM Laboratory Vienna, Techn. Report TR 25.097, 30 June 1969.

/5/ WALK, K., ALBER, K., FLECK, M., GOLDMANN, H., LAUER, P., MOSER, E., OLIVA, P., STIGLEITNER, H., ZEISEL, G.: Abstract Syntax and Interpretation of PL/I (ULD Version III). IBM Laboratory Vienna, Techn. Report TR 25.098, 30 April 1969

/6/ ALBER, K., GOLDMANN, H., LAUER, P., LUCAS, P., OLIVA, P., STIGLEITNER, H., WALK, K., ZEISEL, G.: Informal Introduction to the Abstract Syntax and Interpretation of PL/I (ULD Version III). IBM Laboratory Vienna, Techn. Report TR 25.099, 30 June 1969.

The method and notation used in these documents are essentially taken over from the first version of a formal definition of PL/I issued by the Vienna Laboratory:

/7/ PL/I Definition Group of the Vienna Laboratory: Formal Definition of PL/I. IBM Laboratory Vienna, Techn. Report TR 25.071, 30 December 1966

/8/ ALBER, K.: Syntactical Description of PL/I Text and its Translation into Abstract Normal Form. IBM Laboratory Vienna, Techn. Report TR 25.074, 14 April 1967.

An outline of the method is given in:

/9/ LUCAS, P., LAUER, P., STIGLEITNER, H.: Method and Notation for the Formal Definition of Programming Languages. IBM Laboratory Vienna, Techn. Report TR 25.087, 28 June 1968.

This document also contains the appropriate references to the relevant literature. The basic ideas and their application to PL/I have been made available through several workshops on the formal definition of PL/I, and presentations and publications inside and outside IBM. The method is demonstrated by application to an appropriately tailored subset of PL/I in:

/10/ LUCAS, P., WALK, K.: On the Formal Description of PL/I. To be published in Annual Review in Automatic Programming - Vol.6. Pergamon Press, New York 1969.

The language defined in the present version is PL/I as specified in the PL/I Language Specifications, Form No. Y33-6003-1, with the addition of attention handling, input stream and string scanning, and file variables.

The present document will be made subject to validation by the PL/I Language Department, Hursley.

## PRODUCTION

This document was prepared by means of automated text-processing systems. TEXT 360 was used for processing the prose parts. The formatting, indexing, cross-referencing, and updating of formula texts was handled by means of FORMULA 360.

FORMULA 360 is a syntax-controlled formula processing system which was developed in the Vienna Laboratory especially to facilitate the production and maintenance of PL/I Formal Definition documents. The achievements of K.F. KOCH in the overall design and implementation of FORMULA 360 are acknowledged in particular. Essential components of the system are due to G. URSCHLER (syntactical decomposition of formulas) and E. MOSER (formula input checker). H. Hoja and G. Zeisel contributed to the clarification and formulation of the required formatting processes.

Coordination: F. Schwarzenberger, M. Stadler

Technical control: K.F. Koch, E. Moser, W. Pachl, M. Stadler

Data transcription: Miss W. Schatzl, Mrs. H. Deim, and sub-contractors

System support: H. Chladek, G. Lehmayer

CONTENTS

# 1. INTRODUCTION

The present paper contains the complete formal definition of the PL/I compile time facilities. This formal definition supplements the formal definition of PL/I as given in the documents of the PL/I-Definition Group of the Vienna Laboratory.

The compile time facilities here are considered to be a higher level language used to modify programs written in another higher level language (namely PL/I), before the compilation of these programs is performed.

Although the compile time facilities are part of PL/I, the modifying "Compile Time Language" (called CT-Language in the following) may conceptually be separated from the modified language. In particular, the CT-Language is almost unaffected by changes to PL/I (i.e., PL/I without compile time facilities), most of its properties being independent of properties of PL/I.

A great number of concepts and characteristics of PL/I, though in a simplified and restricted form, are also present in the CT-Language. Some other concepts, of course, are strictly oriented towards program modification, as e.g., the scanning and replacement mechanism for generating the output of a compile time program execution.

The principal methods and the notation for the formal definition of the CT-Language have been taken from "Method and Notation for the Formal Definition of Programming Languages" /9/ and from the Formal Definition of of PL/I /3,4,5/. The resulting mechanisms in many respects are simpler than those defining PL/I, but the various new concepts of the CT-Language required the introduction of new parts, not to be found in the formalization of PL/I. It is assumed that the reader is familiar with the methods and notations given in the Formal Definition of PL/I /3,4,5/ but the study of the formal definition of PL/I itself is not necessary.

The complete definition of PL/I (including compile time facilities) now is achieved by a two step mechanism, working on a concrete PL/I program including compile time statements:

(a) The concrete text is considered to be a program written in the CT-Language. The formal definition of the CT-Language as found in the present document is used for the interpretation of that ct-program. The outcome of the interpretation is a list of character values to be found in the result cell (see chapters 8,9).

(b) The resulting text of step (a) is considered to be a concrete PL/I program without compile time facilities, and the formal mechanisms for PL/I, as defined in the PL/I Translator /4/ and the PL/I Interpreter /5/ are applied.

As for PL/I, the whole defining process for the CT-Language is partitioned into a number of sequentially applicable steps.

(1) The concrete syntax (chapter 3) defines a class of concrete ct-programs by a set of formal production rules. The rules are written in an extended Backus notation as defined in /3/.

(2) In order to remain within the range of methods and concepts used throughout the formal definition, a concrete compile time program which is a string of concrete PL/I characters is transcribed into a list of character values, i.e., of abstract elementary objects representing uniquely the concrete PL/I characters.

(3) The list of character values (representing a concrete compile time program) is mapped by the function parse onto a structured object, called the "abstract representation" of the concrete compile time program. (This object may be seen as an abstract form of the parsing tree of the concrete

program). The structure of these objects is described in chapter 5 by a set of predicate definitions, called "abstract representation of the concrete syntax".

(4) In chapter 6 the function translate is specified, which translates the abstract representation of a concrete ct-program into an "abstract ct-program". The main task is the recognition of all declarations in the concrete ct-program and the test for multiple declarations. For the other components of a concrete ct-program, the translation consists essentially of a one-to-one mapping from the "parsing-tree" into the abstract ct-program. The structure of an abstract ct-program is described in chapter 7 by a set of predicate definitions, called the "abstract syntax". (Although the specification of the abstract syntax is redundant - because it is given implicitely by the function translate - it is not only a great help towards the intelligibility of the formal definition but also very useful for the treatment of language questions of theoretical nature.)

(5) In chapter 9 the formal definition of the interpreter is given by the definition of an abstract machine which is characterized by the set of its possible states and its state transitions. The behaviour of the abstract machine which in its initial state contains an abstract compile time program defines the interpretation of that abstract ct-program.

The above described concept of partitioning the definition process has consequences for the way in which invalid ct-programs are rejected. The concrete syntax defines a class of concrete ct-programs including, of course, programs which have no interpretation. To some degree, the restrictiveness of the syntax is arbitrary. A certain class of syntactically correct programs are discarded by the translator. However, in order to make the abstract syntax together with the interpreter to a self-contained system, the interpreter relies only on the correctness of the abstract programs it interprets according to the abstract syntax. This means that no use is made in the interpreter of the fact that certain programs, although correct according to the abstract syntax, could not have resulted from the translator.

The exclusion of invalid programs by both the translator and the interpreter in most cases is performed by conditional expressions occurring in the definition with an alternative "proposition -→ error" (in function definitions) or "proposition -→ error" (in instruction definitions) for the invalid cases.

A computation of an abstract ct-program is successfully terminated only if the control component of the last entered machine state contains the special "object" Ω. After a successfully terminated computation the result component of the machine state holds a list of character values, the outcome of the interpretation, which is to be considered a concrete PL/I program without compile time facilities.

## 2. NOTATION AND CONVENTIONS

Throughout the present document the notation and conventions introduced in

section 2.1 and 2.2 of /3/ and

chapter 1 of /5/

are used without any special reference. However, the following sections constitute a slight modification of some parts of chapter 1 of /5/.

### 2.1 THE CLASS OF OBJECTS

The class of objects used in the present document is characterized by the class of simple selectors and the class of elementary objects.

The following simple selectors are used:

(1) Selectors denoted by strings of small letters, digits, and hyphens, prefixed by s-

(2) The range of the following functions:

| | |
|---|---|
| s(i) | for is-intg-val(i) |
| elem(i) | for is-intg-val(i) |
| mk-id(identifier) | for is-identifier(identifier) |
| af(scope,id) | for (is-* ∨ is-ad)(scope) & is-id(id) |
| sel(idp) | for is-id-pair(idp) |

The ranges of these one to one functions are mutually disjoint.

(3) The infinite class $SUCC_0$ of simple selectors which are used to select the immediate sub-trees of a control tree (cf. 1.3.2 of /5/).

From these selectors the semigroup of all composite selectors is formed, including the identity function I, which is the unity with respect to functional composition.

The following classes of elementary objects are used, which are mutually disjoint:

(1) Objects denoted by strings of capital letters, hyphens, and digits, without definition place, and the special objects *, <>, ¤.

(2) Integer values, satisfying the predicate is-intg-val.

(3) Character values, satisfying the predicate is-char-val, but not mentioned under (1) (extralingual characters).

(4) Composite selectors.

(5) Finite sets of objects and the empty set {}.

## 2.2 BASIC FUNCTIONS AND PREDICATES

This section defines all functions whose ranges are simple selectors, as mentioned in the foregoing section, and some predicates over elementary objects.

(1) s(i) =

for:is-intg-val(i)

(2) is-intg-val(i1) & is-intg-val(i2) & i1 ≠ i2 ⊃ s(i1) ≠ s(i2)

(3) elem(i) =

for:is-intg-val(i)

Note: This function maps integer values into simple selectors, called 'unique names'. It is partially described by the both following axioms.

(4) is-intg-val(i1) & is-intg-val(i2) & i1 ≠ i2 ⊃ elem(i1) ≠ elem(i2)

(5) is-intg-val(i) ⊃ is-n•elem(i)

(6) is-n =

Note: This predicate characterizes the range of the function elem.

(7) is-pointer =

Note: This predicate describes the subclass of composite selectors, which is composed of simple selectors of the classes elem(i) and s(i). The identity function I is included.

(8) mk-id(cvl) =

for:is-identifier(cvl)

Ref.: is-identifier 9-36(144)

Note: This function maps special lists of character values, namely identifiers, into 'abstract identifiers'. The function is partially described by the both following axioms.

In some applications of the function mk-id a shorthand notation is used for the argument, e.g.,

mk-id(INDEX) stands for mk-id(<I-CHAR,N-CHAR,D-CHAR,E-CHAR,X-CHAR>).

(9) is-identifier(cv11) & is-identifier(cv12) & cv11 ≠ cv12 ⊃
mk-id(cv11) ≠ mk-id(cv12)

Ref.: is-identifier 9-36(144)

(10) is-identifier(cv1) ⊃ is-id•mk-id(cv1)

Ref.: is-identifier 9-36(144)

(11) is-id =

Note: This predicate characterizes the range of the function mk-id.

(12) af(scope,id) =

for:(is-* ∨ is-ad)(scope) & is-id(id)

Note: This 'address function' maps an abstract identifier together with its scope information onto an 'address' under which the denotation of the identifier is stored in the denotation directory of the CT-Machine. The scope information indicates whether the identifier is global in the program under consideration (*), or is local to a procedure body, in which case the address of the body is used as scope information. This one to one function insures the static storage class of ct-variables.

(13) (is-* ∨ is-ad)(scope1) & (is-* ∨ is-ad)(scope2) & is-id(id1) & is-id(id2) &
af(scope1,id1) = af(scope2,id2) ⊃ scope1 = scope2 & id1 = id2

(14) (is-* ∨ is-ad)(scope) & is-id(id) ⊃ is-ad•af(scope,id)

(15) is-ad =

Note: This predicate characterizes the range of the function af.

(16) sel(idp) =

for:is-id-pair(idp)

Note: This function maps a pair of identifiers onto a selector.

(17) is-id-pair(idp1) & is-id-pair(idp2) & idp1 ≠ idp2 ⊃ sel(idp1) ≠ sel(idp2)

(18) is-intg-val =

Note: This predicate characterizes the class of all (positive, zero, and negative) integer values.

## 2.3 REFERENCING

All function and instruction names occurring in a formula are referenced under the heading 'Ref.:', where a reference has the following notation:

name chapter - page (formula-number)

There are the following exceptions to this rule:

(1) All names defined in chapter 1 of /5/ are not referenced.

(2) All names defined in chapter 2 are not referenced.

(3) All names defined in the abstract representation of concrete syntax (chapter 5) and abstract syntax (chapter 7) are not referenced.

(4) All names defined in a sub-chapter are not referenced throughout this sub-chapter.

# 3. CONCRETE SYNTAX

This chapter contains the syntactic description of concrete PL/I compile time programs, called the "concrete syntax".

The concrete syntax is given by a set of formal production rules, written in a modified Backus notation. The syntax and semantics of this notation is defined in "Concrete Syntax of PL/I" /3/.

The generation process for a concrete compile time program, as well as the rules for the 48 character set version, slightly differ from the corresponding sections of /3/ and hence are also given below.

## 3.1 GENERATION OF A CONCRETE PROGRAM

### 3.1.1 THE NORMAL GENERATION PROCESS

Since PL/I has context dependent rules for the insertion of blanks and comments, which cannot be expressed by production rules of the modified Backus form, the generation of a concrete compile time program will be performed in four steps:

(1) Starting with the notation variable "program" replacements are to be performed by using the higher level production rules listed in 3.2.1. The process is terminated if none of the higher level production rules is further applicable. The resulting text consists of "ct-words" which are listed in 3.2.3.

(2) Now, "spaces" are inserted into the generated text according the following rule:

The 21 ct-words

= + - * / ( ) , ; : & | ¬ > < || >= <= ¬> ¬= ¬<

are "delimiters". All other ct-words with exception of the notation variable "text" are "non-delimiters". Between two adjacent non-delimiters the notation variable "space" must be inserted. Preceding and following the notation variable "text" the insertion of "space" is not allowed. Between all other combinations of ct-words the notation variable "space" may be inserted.

With the production rule

space ::= { blank | comment }•••

replacements are to be performed.

(3) The replacement is continued by application of the lower level production rules listed in 3.2.2 and the implementation dependent lower level production rule for the notation variable "extralingual-character".

(4) Finally all notation constants are split up into their single symbols. The generation ends up with a text consisting of the 60 character PL/I alphabet:

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 _ blank = + - * / ( ) , . ; : ' & | ¬ > < ? %

and the implementation dependent extralingual characters.

### 3.1.2 KEYWORD ABBREVIATIONS

The compile time facilities contain the possibility of using keyword-abbreviations. This facilty is not given in section 3.2 because it would lengthen unnecessarily the production rules.

The following abbreviations may be inserted instead of the corresponding keywords. The necessary replacement must be done before step 3 of the generation process takes place.

| | |
|---|---|
| ACTIVATE | ACT |
| CHARACTER | CHAR |
| DEACTIVATE | DEACT |
| DECLARE | DCL |
| PROCEDURE | PROC |

### 3.1.3 PROGRAMS IN THE 48 CHARACTER SET

The alternative use of a restricted character set for writing compile time programs is possible. The character set consists of the following 48 characters:

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 blank = + - * / ( ) . , '

To generate a program in the 48 character set in addition to the process described above the following rules have to be obeyed:

(1) 14 ct-words have to be interpreted as notation variables. They must be replaced by means of the following higher level production rules during step (1):

| | |
|---|---|
| : ::= .. | < ::= LT |
| ; ::= ,. | >= ::= GE |
| % ::= // | <= ::= LE |
| ¬ ::= NOT | ¬> ::= NG |
| & ::= AND | ¬< ::= NL |
| ⊥ ::= OR | ¬= ::= NE |
| > ::= GT | ⊥⊥ ::= CAT |

For the insertion of spaces the ct-words ".." and ",." are handled as delimiters and the other resulting ct-words as non-delimiters.

(2) From the lower production rules (3.2.2) for

letter
alphameric-character
string-character
text-character
string-part-char
comment-symbol

the following 12 symbols have to be deleted

@ # - ; : & ⊥ ¬ > < ? %

(3) The 11 sequences of letters

NOT, AND, OR, GT, LT, GE, LE, NG, NL, NE, CAT

are "reserved words", i.e., no notation variable "identifier" must finally be replaced by any of them.

(4) The lower level production rule for text has to be replaced by

text ::=

[ *••• ] | /••• | { [ *••• | /••• ] { text-character | string-part } |

[ *••• | / ] { blank | comment } }••• [ *••• | /••• ]

where from the production rule for text-character the character "blank" is deleted.

Note: By that way the terminal strings of text do not contain substrings with the structure //space outside string-parts and comments.

## 3.2 PRODUCTION RULES

### 3.2.1 HIGHER LEVEL PRODUCTION RULES

(1) program ::=

text [ { sentence text }••• ]

(2) sentence ::=

statement | declare-statement | procedure

(3) statement ::=

% [ labellist ] { if-statement | unconditional-statement }

(4) labellist ::=

{ identifier : }•••

(5) if-statement ::=

if-clause statement | if-clause balanced-statement % ELSE statement

(6) if-clause ::=

IF expression % THEN

(7) balanced-statement ::=

% [ labellist ] { if-clause balanced-statement % ELSE balanced-statement | unconditional-statement }

(8) unconditional-statement ::=

group | goto-statement | include-statement | assignment-statement | null-statement | activate-statement | deactivate-statement

(9) group ::=

simple-group | iterated-group

(10) simple-group ::=

DO ; program end-clause

(11) end-clause ::=

% [ labellist ] END [ identifier ] ;

(12) iterated-group ::=

DO do-specification ; program end-clause

(13) do-specification ::=

identifier = expression [ BY expression [ TO expression ] |
TO expression [ BY expression ] ]

(14) goto-statement ::=

{ GOTO | GO TO } identifier ;

(15) include-statement ::=

INCLUDE { , • library-specification••• } ;

(16) library-specification ::=

[ identifier ] ( identifier ) | identifier

(17) assignment-statement ::=

identifier = expression ;

(18) expression ::=

expression-six | expression ⊥ expression-six

(19) expression-six ::=

expression-five | expression-six & expression-five

(20) expression-five ::=

expression-four | expression-five comparison-operator expression-four

(21) comparison-operator ::=

> | >= | = | < | <= | ¬> | ¬= | ¬<

(22) expression-four ::=

expression-three | expression-four || expression-three

(23) expression-three ::=

expression-two | expression-three { + | - } expression-two

(24) expression-two ::=

expression-one | expression-two { * | / } expression-one

(25) expression-one ::=

primitive-expression | { + | - | ¬ } expression-one

(26) primitive-expression ::=

( expression ) | reference | constant

(27) reference ::=

identifier [ ( { , ∘ expression∘∘∘ } ) ]

(28) null-statement ::=

;

(29) activate-statement ::=

ACTIVATE { , ∘ activation∘∘∘ } ;

(30) activation ::=

identifier [ RESCAN | NORESCAN ]

(31) deactivate-statement ::=

DEACTIVATE { , ∘ identifier∘∘∘ } ;

(32) declare-statement ::=

% [ labellist ] DECLARE { , ∘ declaration∘∘∘ } ;

(33) declaration ::=

[ identifier | ( [ , ∘ identifier∘∘∘ } ) } attribute

(34) attribute ::=

CHARACTER | FIXED | ENTRY

(35) procedure ::=

% labellist PROCEDURE [ parameterlist ] RETURNS ( { CHARACTER | FIXED } ) ; [ p-sentence••• ] end-clause

(36) parameterlist ::=

( { , • identifier••• } )

(37) p-sentence ::=

p-statement | p-declare-statement

(38) p-statement ::=

[ labellist ] { p-if-statement | p-unconditional-statement }

(39) p-if-statement ::=

p-if-clause p-statement | p-if-clause p-balanced-statement ELSE p-statement

(40) p-if-clause ::=

IF expression THEN

(41) p-balanced-statement ::=

[ labellist ] { p-if-clause p-balanced-statement ELSE p-balanced-statement | p-unconditional-statement }

(42) p-unconditional-statement ::=

p-group | goto-statement | assignment-statement | null-statement | return-statement

(43) p-group ::=

p-simple-group | p-iterated-group

(44) p-simple-group ::=

DO ; [ p-sentence••• ] p-end-clause

(45) p-end-clause ::=

[ labellist ] END [ identifier ] ;

(46) p-iterated-group ::=

DO do-specification ; [ p-sentence••• ] p-end-clause

(47) return-statement ::=

RETURN ( expression ) ;

(48) p-declare-statement ::=

[ labellist ] DECLARE { , • p-declaration••• } ;

(49) p-declaration ::=

{ identifier | ( { , • identifier••• } ) } p-attribute

(50) p-attribute ::=

CHARACTER | FIXED | BUILTIN

### 3.2.2 LOWER LEVEL PRODUCTION RULES

(51) identifier ::=

letter [ alphameric-character••• ]

(52) 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 | $ | @ | #

(53) alphameric-character ::=

letter | digit | _

(54) digit ::=

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(55) constant ::=

integer | character-string | bit-string

(56) integer ::=

digit•••

(57) character-string ::=

' [ string-character••• ] '

(58) string-character ::=

alphameric-character | blank | '' | = | + | - | * | / |
( | ) | , | . | ; | : | & | ⊥ | ¬ | > | < | ? | % | extralingual-character

(59) bit-string ::=

' [ bit••• ] ' B

(60) bit ::=

0 | 1

(61) text ::=

[ *••• ] | / ••• | { [ *••• | / ••• ] { text-character | string-part |
comment } }••• [ *••• | / ••• ]

(62) text-character ::=

alphameric-character | blank | = | + | - | ( | ) | , | . |
; | : | & | ⊥ | ¬ | > | < | ?

(63) string-part ::=

' [ string-part-char••• ] '

(64) string-part-char ::=

alphameric-character | blank | = | + | - | * | / | ( | ) |
, | . | ; | : | ' | & | ⊥ | ¬ | > | < | ? | % | extralingual-character

(65) comment ::=

/ * [ ( [ *••• ] comment-symbol | / )••• ] *••• /

(66) comment-symbol ::=

alphameric-character | blank | = | + | - | ( | ) | , | . |
; | : | ' | & | ⊥ | ¬ | > | < | ? | % | extralingual-character

(67) extralingual-character ::=

Note: This production is implementation defined.

3.2.3 List of CT-words

```
<
<=
(
+
|
||
&
*
)
;
¬
¬<
¬>
¬=
-
/
,
%
>
>=
:
=
ACTIVATE
BUILTIN
BY
CHARACTER
constant
DEACTIVATE
DECLARE
DO
ELSE
END
ENTRY
FIXED
GO
GOTO
identifier
IF
INCLUDE
NORESCAN
PROCEDURE
RESCAN
RETURN
RETURNS
text
THEN
TO
```

### 3.2.4 CROSS-REFERENCE INDEX

This index lists all terminals and non-terminals of the concrete syntax. For all names all instances of use in a production are given. For non-terminals the defining production is indicated by an underlined reference. A reference has the form 3-yy(zz), where yy is the page number within chapter 3 and zz is the number of the production.

## 4. CORRESPONDENCE BETWEEN A CONCRETE PROGRAM AND ITS ABSTRACT REPRESENTATION

The "abstract representation" of a concrete program is an object, whose structure represents the syntactical structure of the concrete program and whose elementary components are character values which correspond to the concrete characters of the concrete program. One may think of the abstract representation as the parsing tree of the concrete program.

The abstract representation of a concrete program satisfies the predicate is-c-program. This predicate is defined in chapter 5 by a set of predicate definitions, called "abstract representation of concrete syntax".

The abstract representation of the concrete syntax may also be considered as a normal form with regard to the two different production systems, corresponding to the 60- and to the 48-character set version. For instance, the relational operator "greater than" is designated in the 60-character set version by the character ">", in the 48-character set version by the character "G", followed by the character "T". In the abstract representation this relation is designated by the elementary object "GT", independently from the concrete representation.

It should be noted that the abstract representation of a concrete program contains solely information relevant for the semantics of the program, i.e., "spaces" (blanks and comments) are not contained. Hence, one abstract represented program corresponds to an infinite set of equivalent concrete programs, which differ in their spaces, according to point 3 of the generation process defined in section 3.1.1.

Each predicate definition out of the abstract representation of the concrete syntax is closely related to a production out of the 60-character set production system. This one to one mapping is described in Appendix I of /4/. (One could map in a similar way the 48-character set production system onto a set of predicate definitions with the head, say is-c-program-48, such that is-c-program-48 ⊃ is-c-program).

In the following two parsing functions are defined, namely "parse" and "parse-48", mapping a concrete program of the 60- and of the 48-character set, respectively, onto its abstract representation. The argument of these functions is not a concrete program itself which is a string of concrete PL/I characters, but rather a representation of the concrete program as a list of elementary objects satisfying the predicate is-char-val. The correspondence between the individual concrete characters and the character values is given in Appendix I of /4/.

No constructive algorithm is given for the parsing functions parse and parse-48, because a special algorithm would lead to a loss of universality without contributing any information with regard to the formal definition. parse and parse-48 are defined as the inverse of the functions generate and generate-48, respectively, providing the unambiguity of the concrete syntax.

In order to avoid accumulation of confusing parenthesis in expressions of the form (s(1))•(s(n))•p(t) the following abbreviation is used throughout chapters 4 and 6:

$$s_n = s(n) \quad \text{for} \quad \text{is-intg-val}(n)$$

Using this convention the expression above reads: $s_1 \bullet s_n \bullet p(t)$.

Metavariables:

is-char-val-list(txt) a concrete program

is-c-program(t) the abstract representation of a concrete prpgram

is-intg-val(i) & i > 0

is-intg-val(j) & j > 0

is-pointer(p)

Abbreviation:

$s_n$ = s(n)

## 4.1 PROGRAMS IN THE 60-CHARACTER SET

(1) parse(txt) =

(∪t)(txt ∈ generate(t))

(2) generate(t) =

{lin-3(x) | x ∈ insert-space•lin-1(t)}

(3) lin-1(t) =

$$text_0\hat{} \overset{n_0}{\underset{n=1}{CONC}} (lin\text{-}2 \bullet s_1 \bullet s_n \bullet s_2(t)\hat{}text_n)$$

where:

$text_0$ = (is-Ω•$s_1$(t) → <ϖ>, T → <ϖ>^<$s_1$(t)>^<ϖ>)

$text_n$ = (is-Ω•$s_2$•$s_n$•$s_2$(t) → <ϖ>, T → <ϖ>^<$s_2$•$s_n$•$s_2$(t)>^<ϖ>)

$n_0$ = slength•$s_2$(t)

Note: The function lin-1 planes t into a list of those components which represent syntactical units that may not be interrupted by spaces (blanks and comments). This is done by planing the structure given by selectors of the class s(n) or s-del, but not affecting the structure given by selectors of the class elem(n). (This limit corresponds in a concrete program to the limit between higher and lower level syntax). Furthermore, the dummy object "ϖ" is inserted between a sentence and a text, or between two sentences, according to the main structure of t (cf. 5-1(2)), indicating where the insertion of space is also forbidden (cf. rule 2 of the generation process, defined in 3.1.1).

(4) slength(x) =

(∀i)(is-Ω•(s(i))(x)) → 0

T → (∪i)(¬is-Ω•(s(i))(x) & (∀j)(j > i ⊃ is-Ω•(s(j))(x)))

Note: This function is defined for any object x. If x has no s(i)-component it yields 0, else the maximum index of an existing s(i)-component.

(5) lin-2(x) =

$is\text{-}\Omega(x) \rightarrow \langle\rangle$

$slength(x) = 0 \rightarrow \langle x \rangle$

$is\text{-}c\text{-}simple\text{-}group(x) \rightarrow \langle s_1(x)\rangle\hat{}\langle s_2(x)\rangle\hat{}lin\text{-}1 \circ s_3(x)\hat{}lin\text{-}2 \circ s_4(x)$

$is\text{-}c\text{-}iterated\text{-}group(x) \rightarrow \langle s_1(x)\rangle\hat{}lin\text{-}2 \circ s_2(x)\hat{}\langle s_3(x)\rangle\hat{}lin\text{-}1 \circ s_4(x)\hat{}lin\text{-}2 \circ s_5(x)$

$T \rightarrow lin\text{-}2 \circ s_1(x)\hat{} \underset{n=2}{\overset{n_0}{CONC}} (lin\text{-}2 \circ s\text{-}del(x)\hat{}lin\text{-}2 \circ s_n(x))$

where:
$n_0 = slength(x)$

Note: Groups outside procedures contain an abstract represented program, hence the function lin-1 must be applied again.

(6) insert-space(x) =

$is\text{-}\langle\rangle(x) \rightarrow \{\langle\rangle\}$

$head(x) = \pi \rightarrow insert\text{-}space \circ tail(x)$

$head \circ tail(x) = \pi \rightarrow \{\langle head(x)\rangle\hat{}z \mid z \in insert\text{-}space \circ tail \circ tail(x)\}$

$is\text{-}c\text{-}delimiter \circ head(x) \vee is\text{-}c\text{-}delimiter \circ head \circ tail(x) \rightarrow$

$\{mklist(head(x), y, z) \mid (is\text{-}c\text{-}space \vee is\text{-}\Omega)(y) \ \& \ z \in insert\text{-}space \circ tail(x)\}$

$T \rightarrow [mklist(head(x), y, z) \mid is\text{-}c\text{-}space(y) \ \& \ z \in insert\text{-}space \circ tail(x)\}$

for: is-list(x)

Note: The function insert-space intersperses its argument, which is a list, with spaces. In general it yields the infinite set of all lists resulting from this interspersion satisfying the condition that at least between all pairs of consecutive non-delimiters spaces are inserted, unless the object "π" occurs, which has to be omitted and no interspersion occurs there.

(7) is-c-delimiter(x) =

$x \in$
{EQ, PLUS, MINUS, ASTER, SLASH, LEFT-PAR, RIGHT-PAR, COMMA, SEMIC, COLON,
AND, OR, NOT, GT, LT, <OR, OR>, <GT, EQ>, <LT, EQ>, <NOT, GT>, <NOT, EQ>, <NOT, LT>}

(8) is-c-space =

(<elem(1):is-BLANK ∨ is-c-comment>, ...)

(9) mklist(a,b,list) =

$\mu_0$(<elem(1):a>,<elem(2):b>)^list

for:¬is-Ω(a) & is-list(list)

(10) lin-3(x) =

is-Ω(x) --> <>

is-char-val(x) --> <x>

$T \dashrightarrow \overset{i_0}{\underset{i=1}{\mathrm{CONC}}}$ lin-3•elem(i,x)

where:
$i_0$ = (∪i)(¬is-Ω•elem(i,x) & (∀j)(j > i ⊃ is-Ω•elem(j,x)))

Note: This function linearizes its argument into a list of elementary objects. This is done by planing the structure given by selectors of the class elem(i).

## 4.2 PROGRAMS IN THE 48-CHARACTER SET

For programs in the 48-character set some additional tests are necessary. On the one hand the context dependent property is proved that the 11 "reserved words" may not be used as identifiers (cf. rule 3 of section 3.1.3), on the other hand, the abstract representation of concrete syntax which is equivalent to the 60-character set production system, must be restricted to reflect the both 48-character set limitations, specified by rules 2 and 4 of section 3.1.3, which are context free.

(11) parse-48(txt) =

(∪t)(txt ∈ generate-48(t))

(12) generate-48(t) =

```
¬(∃p)(is-c-identifier•p(t) &
   (is-c-NOT•lin-3•p(t) ∨ is-c-AND•lin-3•p(t) ∨ is-c-OR•lin-3•p(t) ∨
      is-c-GT•lin-3•p(t) ∨ is-c-LT•lin-3•p(t) ∨ is-c-GE•lin-3•p(t) ∨
      is-c-LE•lin-3•p(t) ∨ is-c-NG•lin-3•p(t) ∨ is-c-NL•lin-3•p(t) ∨
      is-c-NE•lin-3•p(t) ∨ is-c-CAT•lin-3•p(t))) -->

  {lin-3(x) | x ∈ insert-space-48•replace-48•lin-1-48(t) &
  ¬(∃i){elem(i,lin-3(x)) ∈
     {COMM-AT,NUMBER-SIGN,BREAK,SEMIC,COLON,AND,OR,NOT,GT,LT,QUEST,PERC})}

T --> error
```

Ref.: lin-3 4-4(10)

Note: cf. points 1 to 3 of section 3.1.3.

(13) lin-1-48(t) =

$$\text{text-48}\bullet s_1(t)\,\hat{}\,\mathop{\text{CONC}}_{n=1}^{n_0}(\text{lin-2-48}\bullet s_1\bullet s_n\bullet s_2(t)\,\hat{}\,\text{text-48}\bullet s_2\bullet s_n\bullet s_2(t))$$

where:
$n_0$ = slength•$s_2$(t)

Ref.: slength 4-2(4)

(14) text-48(x) =

is-Ω(x) → <ʷ>

¬(∃i)(is-SLASH•elem(1,$el_1$) & is-SLASH•elem(2,$el_1$) &
(is-BLANK ∨ is-c-comment)($el_2$)) →

<ʷ>^<x>^<ʷ>

T → error

where:
$el_1$ = (elem(1))•(elem(i))•elem(1,x)
$el_2$ = (elem(2))•(elem(i))•elem(1,x)

for:is-c-text(x)

Note: The restriction on text is necessary for reasons of unambiguity; cf. point 4 of section 3.1.3.

(15) lin-2-48(x) =

is-Ω(x) → <>

slength(x) = 0 → <x>

is-c-simple-group(x) → <$s_1$(x)>^<$s_2$(x)>^lin-1-48•$s_3$(x)^lin-2-48•$s_4$(x)

is-c-iterated-group(x) →

<$s_1$(x)>^lin-2-48•$s_2$(x)^<$s_3$(x)>^lin-1-48•$s_4$(x)^lin-2-48•$s_5$(x)

$$T \rightarrow \text{lin-2-48}\bullet s_1(x)\,\hat{}\,\mathop{\text{CONC}}_{n=2}^{n_0}(\text{lin-2-48}\bullet\text{s-del}(x)\,\hat{}\,\text{lin-2-48}\bullet s_n(x))$$

where:
$n_0$ = slength(x)

Ref.: slength 4-2(4)

(16) replace-48(x) =

$$\underset{i=1}{\overset{i_0}{\mathrm{LIST}}}\ \text{replace-48-1•elem(i,x)}$$

where:
$i_0$ = length(x)

(17) replace-48-1(x) =

is-COLON(x) --→ <POINT,POINT>

is-SEMIC(x) --→ <COMMA,POINT>

is-PERC(x) --→ <SLASH,SLASH>

is-NOT(x) --→ <N-CHAR,O-CHAR,T-CHAR>

is-AND(x) --→ <A-CHAR,N-CHAR,D-CHAR>

is-OR(x) --→ <O-CHAR,R-CHAR>

is-GT(x) --→ <G-CHAR,T-CHAR>

is-LT(x) --→ <L-CHAR,T-CHAR>

x = <GT,EQ> --→ <G-CHAR,E-CHAR>

x = <LT,EQ> --→ <L-CHAR,E-CHAR>

x = <NOT,GT> --→ <N-CHAR,G-CHAR>

x = <NOT,LT> --→ <N-CHAR,L-CHAR>

x = <NOT,EQ> --→ <N-CHAR,E-CHAR>

x = <OR,OR> --→ <C-CHAR,A-CHAR,T-CHAR>

T --→ x

(18) insert-space-48(x) =

is-<>(x) --→ {<>}

head(x) = ¤ --→ insert-space-48•tail(x)

head•tail(x) = ¤ --→ {<head(x)>^z | z ∈ insert-space-48•tail•tail(x)}

is-c-delimiter-48•head(x) ∨ is-c-delimiter-48•head•tail(x) --→

{mklist(head(x),y,z) | (is-c-space ∨ is-Ω)(y) & z ∈ insert-space-48•tail(x)}

T --→ {mklist(head(x),y,z) | is-c-space(y) & z ∈ insert-space-48•tail(x)}

for:is-list(x)

Ref.: mklist 4-4(9)
is-c-space 4-3(8)

(19) is-c-delimiter-48(x) =

is-c-delimiter(x) ∨ x = <POINT,POINT> ∨ x = <COMMA,POINT>

Ref.: is-c-delimiter 4-3(7)

Note: cf. point 1 of section 3.1.3.

## 4.3 KEYWORD ABBREVIATIONS

To include the possibility of abbreviating keywords one has to replace the predicates is-c-[name] (5-1(1)), where [name] is one of the concrete keywords listed in the following table, throughout the sections 5 and 6 by the corresponding predicates is-c-abbr-[name] defined by:

(20) is-c-abbr-[name] =

is-c-[name] ∨ is-c-[abbr-name]

where: [name] and [abbr-name] are pairs of names given by the following table

| [name] | [abbr-name] |
|---|---|
| ACTIVATE | ACT |
| CHARACTER | CHAR |
| DEACTIVATE | DEACT |
| DECLARE | DCL |
| PROCEDURE | PROC |

# 5. ABSTRACT REPRESENTATION OF CONCRETE SYNTAX

## 5.1 PREDICATE DEFINITIONS

The abstract representation of the concrete syntax is a set of predicate definitions which together define the predicate is-c-program. Objects satisfying this predicate are valid arguments for the function translate defined in chapter 6, and for the function generate defined in section 4.1.

The abstract representation of the concrete syntax is closely related with the production rules of the concrete syntax defined in section 3.2.

Predicates of the form is-c-[name] where [name] is a string of capital letters occurring in the sequel are defined by the following scheme:

(1) is-c-[name] =

(<elem(1):is-char$_1$>,...,
<elem(n):is-char$_n$>)

where: char$_1$,...,char$_n$ denote the character values corresponding tc the concrete characters of the string forming [name], and n (n>1) is the length of the string.

Note: Example: is-c-IF =
(<elem(1):is-I-CHAR>,
<elem(2):is-F-CHAR>)

(2) is-c-program =

(<s(1):is-c-text>,
<s(2):is-Ω ∨
(<s(1):(<s(1):is-c-sentence>,
<s(2):is-c-text>)>,...)>)

(3) is-c-sentence =

is-c-statement ∨ is-c-declare-statement ∨ is-c-procedure

(4) is-c-statement =

(<s(1):is-PERC>,
<s(2):is-Ω ∨ is-c-labellist>,
<s(3):is-c-if-statement ∨ is-c-unconditional-statement>)

(5) is-c-labellist =

(<s(1):(<s(1):is-c-identifier>,
<s(2):is-COLON>)>,...)

(6) is-c-if-statement =

(<s(1):is-c-if-clause>,
<s(2):is-c-statement>) ∨
(<s(1):is-c-if-clause>,
<s(2):is-c-balanced-statement>,
<s(3):is-PERC>,
<s(4):is-c-ELSE>,
<s(5):is-c-statement>)

(7) is-c-if-clause =

(<s(1):is-c-IF>,
<s(2):is-c-expression>,
<s(3):is-PERC>,
<s(4):is-c-THEN>)

(8) is-c-balanced-statement =

(<s(1):is-PERC>,
<s(2):is-Ω ∨ is-c-labellist>,
<s(3):(<s(1):is-c-if-clause>,
<s(2):is-c-balanced-statement>,
<s(3):is-PERC>,
<s(4):is-c-ELSE>,
<s(5):is-c-balanced-statement>) ∨ is-c-unconditional-statement>)

(9) is-c-unconditional-statement =

is-c-group ∨ is-c-goto-statement ∨ is-c-include-statement ∨
is-c-assignment-statement ∨ is-c-null-statement ∨ is-c-activate-statement ∨
is-c-deactivate-statement

(10) is-c-group =

is-c-simple-group ∨ is-c-iterated-group

(11) is-c-simple-group =

(<s(1):is-c-DO>,
<s(2):is-SEMIC>,
<s(3):is-c-program>,
<s(4):is-c-end-clause>)

(12) is-c-end-clause =

(<s(1):is-PERC>,
<s(2):is-Ω ∨ is-c-labellist>,
<s(3):is-c-END>,
<s(4):is-Ω ∨ is-c-identifier>,
<s(5):is-SEMIC>)

(13) is-c-iterated-group =

(<s(1):is-c-DO>,
<s(2):is-c-do-specification>,
<s(3):is-SEMIC>,
<s(4):is-c-program>,
<s(5):is-c-end-clause>)

```
(14)   is-c-do-specification =

          (<s(1):is-c-identifier>,
           <s(2):is-EQ>,
           <s(3):is-c-expression>,
           <s(4):is-Ω ∨
                 (<s(1):is-c-BY>,
                  <s(2):is-c-expression>,
                  <s(3):is-Ω ∨
                        (<s(1):is-c-TO>,
                         <s(2):is-c-expression>)>) ∨
                 (<s(1):is-c-TO>,
                  <s(2):is-c-expression>,
                  <s(3):is-Ω ∨
                        (<s(1):is-c-BY>,
                         <s(2):is-c-expression>)>)>)


(15)   is-c-goto-statement =

          (<s(1):is-c-GOTO ∨
                 (<s(1):is-c-GO>,
                  <s(2):is-c-TO>)>,
           <s(2):is-c-identifier>,
           <s(3):is-SEMIC>)


(16)   is-c-include-statement =

          (<s(1):is-c-INCLUDE>,
           <s(2):(<s-del:is-COMMA>,
                  <s(1):is-c-library-specification>,...)>,
           <s(3):is-SEMIC>)


(17)   is-c-library-specification =

          (<s(1):is-Ω ∨ is-c-identifier>,
           <s(2):is-LEFT-PAR>,
           <s(3):is-c-identifier>,
           <s(4):is-RIGHT-PAR>) ∨ is-c-identifier


(18)   is-c-assignment-statement =

          (<s(1):is-c-identifier>,
           <s(2):is-EQ>,
           <s(3):is-c-expression>,
           <s(4):is-SEMIC>)


(19)   is-c-expression =

          is-c-expression-six ∨
          (<s(1):is-c-expression>,
           <s(2):is-OR>,
           <s(3):is-c-expression-six>)


(20)   is-c-expression-six =

          is-c-expression-five ∨
          (<s(1):is-c-expression-six>,
           <s(2):is-AND>,
           <s(3):is-c-expression-five>)
```

```
(21)   is-c-expression-five =

          is-c-expression-four ∨
          (<s(1):is-c-expression-five>,
           <s(2):is-c-comparison-operator>,
           <s(3):is-c-expression-four>)


(22)   is-c-comparison-operator =

          is-GT ∨ is-EQ ∨ is-LT ∨
          (<elem(1):is-GT>,
           <elem(2):is-EQ>) ∨
          (<elem(1):is-LT>,
           <elem(2):is-EQ>) ∨
          (<elem(1):is-NOT>,
           <elem(2):is-GT>) ∨
          (<elem(1):is-NOT>,
           <elem(2):is-EQ>) ∨
          (<elem(1):is-NOT>,
           <elem(2):is-LT>)


(23)   is-c-expression-four =

          is-c-expression-three ∨
          (<s(1):is-c-expression-four>,
           <s(2):(<elem(1):is-OR>,
                  <elem(2):is-OR>)>,
           <s(3):is-c-expression-three>)


(24)   is-c-expression-three =

          is-c-expression-two ∨
          (<s(1):is-c-expression-three>,
           <s(2):is-PLUS ∨ is-MINUS>,
           <s(3):is-c-expression-two>)


(25)   is-c-expression-two =

          is-c-expression-one ∨
          (<s(1):is-c-expression-two>,
           <s(2):is-ASTER ∨ is-SLASH>,
           <s(3):is-c-expression-one>)


(26)   is-c-expression-one =

          is-c-primitive-expression ∨
          (<s(1):is-PLUS ∨ is-MINUS ∨ is-NOT>,
           <s(2):is-c-expression-one>)


(27)   is-c-primitive-expression =

          (<s(1):is-LEFT-PAR>,
           <s(2):is-c-expression>,
           <s(3):is-RIGHT-PAR>) ∨ is-c-reference ∨ is-c-constant
```

(28) is-c-reference =

```
(<s(1):is-c-identifier>,
 <s(2):is-Ω ∨
       (<s(1):is-LEFT-PAR>,
        <s(2):(<s-del:is-COMMA>,
               <s(1):is-c-expression>,...)>,
        <s(3):is-RIGHT-PAR>)>)
```

(29) is-c-null-statement =

```
is-SEMIC
```

(30) is-c-activate-statement =

```
(<s(1):is-c-ACTIVATE>,
 <s(2):(<s-del:is-COMMA>,
        <s(1):is-c-activation>,...)>,
 <s(3):is-SEMIC>)
```

(31) is-c-activation =

```
(<s(1):is-c-identifier>,
 <s(2):is-Ω ∨ is-c-RESCAN ∨ is-c-NORESCAN>)
```

(32) is-c-deactivate-statement =

```
(<s(1):is-c-DEACTIVATE>,
 <s(2):(<s-del:is-COMMA>,
        <s(1):is-c-identifier>,...)>,
 <s(3):is-SEMIC>)
```

(33) is-c-declare-statement =

```
(<s(1):is-PERC>,
 <s(2):is-Ω ∨ is-c-labellist>,
 <s(3):is-c-DECLARE>,
 <s(4):(<s-del:is-COMMA>,
        <s(1):is-c-declaration>,...)>,
 <s(5):is-SEMIC>)
```

(34) is-c-declaration =

```
(<s(1):is-c-identifier ∨
       (<s(1):is-LEFT-PAR>,
        <s(2):(<s-del:is-COMMA>,
               <s(1):is-c-identifier>,...)>,
        <s(3):is-RIGHT-PAR>)>,
 <s(2):is-c-attribute>)
```

(35) is-c-attribute =

```
is-c-CHARACTER ∨ is-c-FIXED ∨ is-c-ENTRY
```

```
(36)   is-c-procedure =

          (<s(1):is-PERC>,
           <s(2):is-c-labellist>,
           <s(3):is-c-PROCEDURE>,
           <s(4):is-Ω ∨ is-c-parameterlist>,
           <s(5):is-c-RETURNS>,
           <s(6):is-LEFT-PAR>,
           <s(7):is-c-CHARACTER ∨ is-c-FIXED>,
           <s(8):is-RIGHT-PAR>,
           <s(9):is-SEMIC>,
           <s(10):is-Ω ∨
                 (<s(1):is-c-p-sentence>,...)>,
           <s(11):is-c-end-clause>)


(37)   is-c-parameterlist =

          (<s(1):is-LEFT-PAR>,
           <s(2):(<s-del:is-COMMA>,
                  <s(1):is-c-identifier>,...)>,
           <s(3):is-RIGHT-PAR>)


(38)   is-c-p-sentence =

          is-c-p-statement ∨ is-c-p-declare-statement


(39)   is-c-p-statement =

          (<s(1):is-Ω ∨ is-c-labellist>,
           <s(2):is-c-p-if-statement ∨ is-c-p-unconditional-statement>)


(40)   is-c-p-if-statement =

          (<s(1):is-c-p-if-clause>,
           <s(2):is-c-p-statement>) ∨
          (<s(1):is-c-p-if-clause>,
           <s(2):is-c-p-balanced-statement>,
           <s(3):is-c-ELSE>,
           <s(4):is-c-p-statement>)


(41)   is-c-p-if-clause =

          (<s(1):is-c-IF>,
           <s(2):is-c-expression>,
           <s(3):is-c-THEN>)


(42)   is-c-p-balanced-statement =

          (<s(1):is-Ω ∨ is-c-labellist>,
           <s(2):(<s(1):is-c-p-if-clause>,
                  <s(2):is-c-p-balanced-statement>,
                  <s(3):is-c-ELSE>,
                  <s(4):is-c-p-balanced-statement>) ∨ is-c-p-unconditional-statement>)


(43)   is-c-p-unconditional-statement =

          is-c-p-group ∨ is-c-goto-statement ∨ is-c-assignment-statement ∨
          is-c-null-statement ∨ is-c-return-statement
```

```
(44)   is-c-p-group =

          is-c-p-simple-group v is-c-p-iterated-group


(45)   is-c-p-simple-group =

          (<s(1):is-c-DO>,
           <s(2):is-SEMIC>,
           <s(3):is-Ω v
                 (<s(1):is-c-p-sentence>,...)>,
           <s(4):is-c-p-end-clause>)


(46)   is-c-p-end-clause =

          (<s(1):is-Ω v is-c-labellist>,
           <s(2):is-c-END>,
           <s(3):is-Ω v is-c-identifier>,
           <s(4):is-SEMIC>)


(47)   is-c-p-iterated-group =

          (<s(1):is-c-DO>,
           <s(2):is-c-do-specification>,
           <s(3):is-SEMIC>,
           <s(4):is-Ω v
                 (<s(1):is-c-p-sentence>,...)>,
           <s(5):is-c-p-end-clause>)


(48)   is-c-return-statement =

          (<s(1):is-c-RETURN>,
           <s(2):is-LEFT-PAR>,
           <s(3):is-c-expression>,
           <s(4):is-RIGHT-PAR>,
           <s(5):is-SEMIC>)


(49)   is-c-p-declare-statement =

          (<s(1):is-Ω v is-c-labellist>,
           <s(2):is-c-DECLARE>,
           <s(3):(<s-del:is-COMMA>,
                  <s(1):is-c-p-declaration>,...)>,
           <s(4):is-SEMIC>)


(50)   is-c-p-declaration =

          (<s(1):is-c-identifier v
                 (<s(1):is-LEFT-PAR>,
                  <s(2):(<s-del:is-COMMA>,
                         <s(1):is-c-identifier>,...)>,
                  <s(3):is-RIGHT-PAR>)>,
           <s(2):is-c-p-attribute>)


(51)   is-c-p-attribute =

          is-c-CHARACTER v is-c-FIXED v is-c-BUILTIN
```

(52) is-c-identifier =

```
(<elem(1):is-c-letter>,
 <elem(2):is-Ω ∨
       (<elem(1):is-c-alphameric-character>,...)>)
```

(53) is-c-letter =

```
is-A-CHAR ∨ is-B-CHAR ∨ is-C-CHAR ∨ is-D-CHAR ∨ is-E-CHAR ∨ is-F-CHAR ∨
is-G-CHAR ∨ is-H-CHAR ∨ is-I-CHAR ∨ is-J-CHAR ∨ is-K-CHAR ∨ is-L-CHAR ∨
is-M-CHAR ∨ is-N-CHAR ∨ is-O-CHAR ∨ is-P-CHAR ∨ is-Q-CHAR ∨ is-R-CHAR ∨
is-S-CHAR ∨ is-T-CHAR ∨ is-U-CHAR ∨ is-V-CHAR ∨ is-W-CHAR ∨ is-X-CHAR ∨
is-Y-CHAR ∨ is-Z-CHAR ∨ is-DOLLAR ∨ is-COMM-AT ∨ is-NUMBER-SIGN
```

(54) is-c-alphameric-character =

```
is-c-letter ∨ is-c-digit ∨ is-BREAK
```

(55) is-c-digit =

```
is-0-CHAR ∨ is-1-CHAR ∨ is-2-CHAR ∨ is-3-CHAR ∨ is-4-CHAR ∨ is-5-CHAR ∨
is-6-CHAR ∨ is-7-CHAR ∨ is-8-CHAR ∨ is-9-CHAR
```

(56) is-c-constant =

```
is-c-integer ∨ is-c-character-string ∨ is-c-bit-string
```

(57) is-c-integer =

```
(<elem(1):is-c-digit>,...)
```

(58) is-c-character-string =

```
(<elem(1):is-APOSTR>,
 <elem(2):is-Ω ∨
       (<elem(1):is-c-string-character>,...)>,
 <elem(3):is-APOSTR>)
```

(59) is-c-string-character =

```
is-c-alphameric-character ∨ is-BLANK ∨
(<elem(1):is-APOSTR>,
 <elem(2):is-APOSTR>) ∨ is-EQ ∨ is-PLUS ∨ is-MINUS ∨ is-ASTER ∨ is-SLASH ∨
is-LEFT-PAR ∨ is-RIGHT-PAR ∨ is-COMMA ∨ is-POINT ∨ is-SEMIC ∨ is-COLON ∨
is-AND ∨ is-OR ∨ is-NOT ∨ is-GT ∨ is-LT ∨ is-QUEST ∨ is-PERC ∨
is-c-extralingual-character
```

(60) is-c-bit-string =

```
(<elem(1):is-APOSTR>,
 <elem(2):is-Ω ∨
       (<elem(1):is-c-bit>,...)>,
 <elem(3):is-APOSTR>,
 <elem(4):is-B-CHAR>)
```

(61) is-c-bit =

```
is-0-CHAR ∨ is-1-CHAR
```

(62) is-c-text =

```
is-Ω ∨
(<elem(1):is-ASTER>,...) ∨
(<elem(1):is-SLASH>,...) ∨
(<elem(1):(<elem(1):(<elem(1):is-Ω ∨
                          (<elem(1):is-ASTER>,...) ∨
                          (<elem(1):is-SLASH>,...)>,
                    <elem(2):is-c-text-character ∨ is-c-string-part ∨
                         is-c-comment>)>,...)>,
 <elem(2):is-Ω ∨
       (<elem(1):is-ASTER>,...) ∨
       (<elem(1):is-SLASH>,...)>)
```

(63) is-c-text-character =

```
is-c-alphameric-character ∨ is-BLANK ∨ is-EQ ∨ is-PLUS ∨ is-MINUS ∨
is-LEFT-PAR ∨ is-RIGHT-PAR ∨ is-COMMA ∨ is-POINT ∨ is-SEMIC ∨ is-COLON ∨
is-AND ∨ is-OR ∨ is-NOT ∨ is-GT ∨ is-LT ∨ is-QUEST
```

(64) is-c-string-part =

```
(<elem(1):is-APOSTR>,
 <elem(2):is-Ω ∨
       (<elem(1):is-c-string-part-char>,...)>,
 <elem(3):is-APOSTR>)
```

(65) is-c-string-part-char =

```
is-c-alphameric-character ∨ is-BLANK ∨ is-EQ ∨ is-PLUS ∨ is-MINUS ∨ is-ASTER ∨
is-SLASH ∨ is-LEFT-PAR ∨ is-RIGHT-PAR ∨ is-COMMA ∨ is-POINT ∨ is-SEMIC ∨
is-COLON ∨ is-AND ∨ is-OR ∨ is-NOT ∨ is-GT ∨ is-LT ∨ is-QUEST ∨ is-PERC ∨
is-c-extralingual-character
```

(66) is-c-comment =

```
(<elem(1):is-SLASH>,
 <elem(2):is-ASTER>,
 <elem(3):is-Ω ∨
       (<elem(1):(<elem(1):is-Ω ∨
                          (<elem(1):is-ASTER>,...)>,
                 <elem(2):is-c-comment-symbol>) ∨ is-SLASH>,...)>,
 <elem(4):(<elem(1):is-ASTER>,...)>,
 <elem(5):is-SLASH>)
```

(67) is-c-comment-symbol =

```
is-c-alphameric-character ∨ is-BLANK ∨ is-EQ ∨ is-PLUS ∨ is-MINUS ∨
is-LEFT-PAR ∨ is-RIGHT-PAR ∨ is-COMMA ∨ is-POINT ∨ is-SEMIC ∨ is-COLON ∨
is-APOSTR ∨ is-AND ∨ is-OR ∨ is-NOT ∨ is-GT ∨ is-LT ∨ is-QUEST ∨ is-PERC ∨
is-c-extralingual-character
```

(68) is-c-extralingual-character =

Note: This predicate is implementation defined. It is equivalent to the predicate is-extralingual-char of the abstract syntax.

## 5.2 CROSS-REFERENCE INDEX

This index lists all names of the abstract representation of the concrete syntax with the exception of the selector generating functions s and elem.

Formulas are referenced by the form 5-yy(zz), where yy is the page number within chapter 5 and zz is the number of the formula. The following conventions hold:

(1) For all names all instances of use in a formula are given. The defining formula is indicated by an underlined reference.

(2) Names of the form is-c-[name], where [name] is a string of capital letters, are defined by the schema given in 5-1(1).

(3) Occurrences of names of the form is-[name], where [name] is a string of capital letters, are listed under the entry [name].

## 6. THE TRANSLATOR

This chapter defines the translation from the abstract representation of a concrete program into an abstract program. This translation is performed by the function translate which maps an object satisfying the predicate is-c-program (cf. 5-1(2)) into an object described by the predicate is-program (cf. 7-1(1)).

All those functions defined in this chapter whose arguments are selectors, possess a further argument which is hidden, namely the abstract represented program to be translated (similar to the machine-state ξ which is a hidden argument of most of the instructions of the interpreter). Throughout this chapter the letter t denotes this hidden argument.

These selectors, called "pointers", are composed of simple selectors of the classes elem(i) and s(i), where i are positive integer values, according to the structure of abstract represented programs.

In this way context dependencies are easily expressible. Even on the translation of parts of t where no context dependency exist, mostly this method is preferred, i.e., a pointer is specified as argument, say p, rather than the component of t to be translated, namely p(t).

Metavariables

```
is-pointer(b) & (is-c-program ∨ is-c-procedure)(b(t))    a pointer to a program or to a procedure

is-pointer(p)

is-pointer(q)

is-pointer(r)

is-intg-val(n) & n > 0
```

Abbreviation:

$s_n = s(n)$

(1) $p \Rightarrow q \equiv (\exists r)(r \neq I \;\&\; q = r \circ p)$

Note: This axiom defines the pointer relation "=>" which is used throughout this chapter. p => q is true iff q is a "continuation" of p. I denotes the unity selector.

(2)
```
translate(t) =

   is-c-program(t) -->

     μ0(<s-decl-part:mk-decl-part(I)>,<s-text-part-list:mk-text-part-list(I)>)

   T --> error
```

Ref.: mk-decl-part 6-2(3)
mk-text-part-list 6-9(29)

Note: I is the unity selector, i.e., the pointer pointing to t itself.

## 6.1 CONSTRUCTION OF THE DECLARATION-PART

This section defines the construction of the declaration-part of the program, as well as the construction of declaration-parts of procedure bodies. These two kinds of declaration-parts,described by the predicates is-decl-part and is-p-decl-part, respectively, differ in that declaration-parts of procedure bodies may not contain declarations of entry names and procedure bodies, but may contain builtin-declarations which in turn may not appear within the declaration-part of the entire program.

The construction of a declaration-part is done in two steps. First, all declarations beeing local to the program or to the procedure under consideration are recognized and the test for multiple declarations is performed. Second, the various types of declarations are constructed and united to the declaration-part.

(3) mk-decl-part(b) =

$\mu_0$({<id:mk-decl(p)> | p $\in$ decl-set(b) & id = mk-id-1•p(t)})

(4) mk-id-1(x) =

mk-id•lin-3(x)

for:is-c-identifier(x)

Ref.: lin-3 4-4(10)

### 6.1.1 RECOGNITION OF DECLARATIONS AND TEST FOR MULTIPLE DECLARATIONS

(5) decl-set(b) =

¬(∃p,q)(is-local-to(b,p) & is-local-to(b,q) & is-mult-decl(p,q)) →

{p | is-decl-cont(p) & is-local-to(b,p) &
(is-entry-cont(p) ⊃ ¬(∃q)(is-entry-decl-cont(q) & q(t) = p(t)))}

T → error

Note: This function yields the set of all pointers pointing to identifiers occurring in the context of declarations beeing local to b(t), minus those pointers pointing to entry names which are also declared by declare statements. An identifier is said to occur in the context of a declaration if it occurs within a declare statement, or in a labellist, or in a list of entry names (also described by the predicate is-c-labellist). Each element of this set becomes a component of the declaration-part. The check for multiple declarations is also performed here. Note, that a procedure together with the possible declarations of its entry names do not constitute a multiple declaration.

(6) is-local-to(b,p) =

$$\text{is-c-procedure}\circ b(t) \dashrightarrow b \Rightarrow p \;\&\; \neg s_2\circ b \Rightarrow p$$

$$\text{is-c-program}\circ b(t) \dashrightarrow$$

$$b \Rightarrow p \;\&\; (\forall q)(\text{is-c-procedure}\circ q(t) \;\&\; b \Rightarrow q \Rightarrow p \supset s_2\circ q \Rightarrow p)$$

(7) is-mult-decl(p,q) =

$$p \neq q \;\&\; \text{is-decl-cont}(p) \;\&\; \text{is-decl-cont}(q) \;\&\; p(t) = q(t) \;\&\; \neg(\text{is-entry-decl-cont}(p) \;\&\; \text{is-entry-cont}(q))$$

(8) is-decl-cont(p) =

$$\text{is-var-decl-cont}(p) \vee \text{is-entry-decl-cont}(p) \vee \text{is-bif-decl-cont}(p) \vee \text{is-entry-cont}(p) \vee \text{is-label-cont}(p)$$

Note: The proposition is true, if the pointer points to an identifier occurring in the context of a declaration. The first 3 alternatives correspond to declare statements, the fourth to the entry-name list of a procedure, the last to a statement-label.

(9) is-var-decl-cont(p) =

$$\text{is-c-identifier}\circ p(t) \;\&\; (\exists q)(\text{is-c-declaration}\circ q(t) \;\&\; q \Rightarrow p \;\&\; (\text{is-c-CHARACTER} \vee \text{is-c-FIXED})(s_2\circ q(t)))$$

(10) is-entry-decl-cont(p) =

$$\text{is-c-identifier}\circ p(t) \;\&\; (\exists q)(\text{is-c-declaration}\circ q(t) \;\&\; q \Rightarrow p \;\&\; \text{is-c-ENTRY}\circ s_2\circ q(t))$$

(11) is-bif-decl-cont(p) =

$$\text{is-c-identifier}\circ p(t) \;\&\; (\exists q)(\text{is-c-p-declaration}\circ q(t) \;\&\; q \Rightarrow p \;\&\; \text{is-c-BUILTIN}\circ s_2\circ q(t))$$

(12) is-entry-cont(p) =

$$\text{is-c-identifier}\circ p(t) \;\&\; (\exists q)(\text{is-c-procedure}\circ q(t) \;\&\; s_2\circ q \Rightarrow p)$$

(13) is-label-cont(p) =

$$\text{is-c-identifier}\circ p(t) \;\&\; (\exists q)((\text{is-c-statement} \vee \text{is-c-declare-statement} \vee \text{is-c-end-clause})(q(t)) \;\&\; s_2\circ q \Rightarrow p \vee (\text{is-c-p-sentence} \vee \text{is-c-p-end-clause})(q(t)) \;\&\; s_1\circ q \Rightarrow p)$$

Note: It can be proved by induction that
is-c-balanced-statement $\supset$ is-c-statement and
is-c-p-balanced-statement $\supset$ is-c-p-statement hold.

### 6.1.2 CONSTRUCTION OF DECLARATIONS

In this section the construction of the four types of declarations is defined. Declarations of variables consist solely either of the elementary object INTG or of the elementary object CHAR. The declaration of an identifier to denote a builtin function consists also of an elementary object, namely BUILTIN. Label and

entry declarations are more complex; they are discussed in the following subsections.

(14) mk-decl(p) =

is-var-decl-cont(p) --> trans-type($attr_1$)

is-bif-decl-cont(p) --> BUILTIN

is-entry-decl-cont(p) --> $\mu_0$(<s-entry-decl:ENTRY>,<s-body:$body_1$>)

is-entry-cont(p) --> $\mu_0$(<s-body:trans-proc(p)>)

is-label-cont(p) --> mk-indexlist(p)

where:
$attr_1$ = ($\iota$x)((∃q)(is-c-declaration•q(t) & q => p & $s_2$•q(t) = x))
$body_1$ = ((∃q)(is-entry-cont(q) & q(t) = p(t)) -->
trans-proc((ιq)(is-entry-cont(q) & q(t) = p(t))),
T --> Ω)

for:is-decl-cont(p)

Note: The declaration of an entry name (cf. formula 5 of chapter 7) is an object with one or two immediate components. In the case that a declare statement exists for the entry name (is-entry-decl-cont), the entry-decl component is the elementary object ENTRY. In the case that a procedure exists for the entry name (is-entry-cont), the body component exists.

(15) trans-type(attr) =

is-c-FIXED(attr) --> INTG

is-c-CHARACTER(attr) --> CHAR

6.1.2.1 Construction of index lists

In this section the declaration of a label, which is an index list, is built up. An index list is a list whose elements are either integer values or truth values, i.e., T or F. The list, read from left to right, represents the location of the labelled statement within the text-part-list of the program (cf. 7-1(1)) or within the procedure-text-part-list of a body (cf. 7-1(6)). Whereby an integer value, say n, indicates that the n-th element of the text-part-list contains the labelled statement; a truth value indicates that the preceding elements of the index list pointed to an if-statement, and that the labelled statement is contained in the then component in the case of T or in the else component in the case of F.

(16) mk-indexlist(p) =

¬(∃b)(is-c-procedure•b(t) & b => p) --> mk-indl-1(p,I)

T --> mk-indl-2(p,$s_{10}$•((ιb)(is-c-procedure•b(t) & b => p)))

for:is-label-cont(p)

Note: Because of the different structures of programs and procedures, a case distinction is necessary, whether the label is local to a procedure or not.

(17) mk-indl-1(p,q) =

$$\text{is-c-labellist}\circ s_2\circ q(t)\ \&\ s_2\circ q \Rightarrow p \rightarrow \langle\rangle$$

$$\text{is-c-if-statement}\circ s_3\circ q(t) \rightarrow \langle s_2\circ s_3\circ q \Rightarrow p\rangle\hat{}\,\text{mk-indl-1}(p,(s(n_1))\circ s_3\circ q)$$

$$\text{is-c-group}\circ s_3\circ q(t) \rightarrow \text{mk-indl-1}(p,(s(n_2))\circ s_3\circ q)$$

$$\neg q \Rightarrow p \rightarrow \langle \text{slength}\circ s_2\circ q(t) + 1 + \text{opt}_1\rangle$$

$$T \rightarrow \langle n_0 + \text{opt}_1\rangle\hat{}\,\text{mk-indl-1}(p,s_1\circ(s(n_0))\circ s_2\circ q)$$

where:

$n_1 = (\iota n)(s_n\circ s_3\circ q \Rightarrow p)$
$n_2 = \text{slength}\circ s_3\circ q(t) - 1$
$\text{opt}_1 = (\text{is-}\Omega\circ s_1\circ q(t) \rightarrow 0,\ T \rightarrow 1)$
$n_0 = (\iota n)(s_n\circ s_2\circ q \Rightarrow p)$

for: is-label-cont(p) &
(is-c-program ∨ is-c-statement ∨ is-c-declare-statement)(p(t))

Ref.: slength 4-2(4)

Note: This function constructs the index list for labels outside procedures.

If a program starts with a text, rather than with a sentence, the first text-part of the text-part-list to be constructed in 6.2 contains a null statement. This fact is taken into account here by the abbreviation $\text{opt}_1$.

The alternative $\neg q \Rightarrow p$ is reached, if the labelled statement is not part of the program (the 3rd or 4th component of a group), but rather is the corresponding end-clause.

Note that the following proposition holds:
is-c-balanced-statement ⊃ is-c-statement.

(18) mk-indl-2(p,q) =

$$\text{is-c-labellist}\circ s_1\circ q(t)\ \&\ s_1\circ q \Rightarrow p \rightarrow \langle\rangle$$

$$\text{is-c-p-if-statement}\circ s_2\circ q(t) \rightarrow \langle s_2\circ s_2\circ q \Rightarrow p\rangle\hat{}\,\text{mk-indl-2}(p,(s(n_1))\circ s_2\circ q)$$

$$\text{is-c-p-group}\circ s_2\circ q(t) \rightarrow \text{mk-indl-2}(p,(s(n_2))\circ s_2\circ q)$$

$$\neg q \Rightarrow p \rightarrow \langle \text{slength}\circ q(t) + 1\rangle$$

$$T \rightarrow \langle n_0\rangle\hat{}\,\text{mk-indl-2}(p,(s(n_0))\circ q)$$

where:

$n_1 = (\iota n)(s_n\circ s_2\circ q \Rightarrow p)$
$n_2 = \text{slength}\circ s_2\circ q(t) - 1$
$n_0 = (\iota n)(s_n\circ q \Rightarrow p)$

for: is-label-cont(p)

Ref.: slength 4-2(4)

Note: This function constructs the index-list for labels which are local to procedures.

The alternative $\neg q \Rightarrow p$ is reached, if the labelled statement is not part of the list of procedure-sentences (in the case of a group), but rather is the corresponding end-clause.

Note that the following proposition holds:
is-c-p-balanced-statement $\supset$ is-c-p-statement.

6.1.2.2 Translation of procedures

Procedures are translated into objects, characterised by the predicate is-body (cf. 7-1(6)). A body has four immediate components: The parameter-list which is a list of identifiers, the return-type which is either INTG or CHAR, the procedure-declaration-part whose construction is already defined in section 6.1, and the procedure-text-part-list whose construction is defined in the sequel.

(19) trans-proc(p) =

$p = en_1 \rightarrow$ mk-body($b_0$)

$T \rightarrow$ mk-id-1$\circ en_1$(t)

where:
$en_1 = s_1 \circ s_1 \circ s_2 \circ b_0$
$b_0 = (\iota b)$(is-c-procedure$\circ$b(t) & b => p)

for:is-entry-cont(p)

Note: Only the first identifier of the entry name list is connected with the body in the declaration-part of the program. All other entry names are associated with the first identifier of the entry name list.

(20) mk-body(b) =

$(\exists n)(s_1 \circ s_n \circ s_2 \circ b(t) = s_4 \circ s_{11} \circ b(t)) \rightarrow$

$\mu_0$(<s-param-list:mk-id-1-list$\circ s_2 \circ s_4 \circ$b(t)>,<s-ret-type:trans-type$\circ s_7 \circ$b(t)>,
<s-decl-part:mk-decl-part(b)>,
<s-text-part-list:trans-p-selist($s_{10} \circ$b)^<$null_0$>>)

$T \rightarrow$ error

where:
$null_0 = \mu_0$(<s-st$\circ$s-st:NULL>)

for:is-c-procedure$\circ$b(t)

(21) mk-id-1-list(slist) =

$$\underset{n=1}{\overset{n_0}{\text{LIST}}}\ \text{mk-id-1}\circ s_n(\text{slist})$$

where:
$n_0$ = slength(slist)

cont'd

Ref.: slength 4-2(4)

(22) trans-p-selist(p) =

$$\overset{n_0}{\underset{n=1}{\mathrm{LIST}}}\ \mu_0(\langle s\text{-}st\text{:}trans\text{-}p\text{-}sentence(s_n \circ p)\rangle)$$

where:
$n_0$ = slength∘p(t)

Ref.: slength 4-2(4)

Note: The similar structures of procedure-text-part-lists and text-part-lists are of advantage on interpreting abstract programs. This is the reason for the s-st component in procedure-text-parts.

(23) trans-p-sentence(p) =

is-c-p-declare-statement∘p(t) → $\mu_0$(<s-st:NULL>)

T → trans-p-st($s_2$∘p)

for:is-c-p-sentence∘p(t)

Note: Concrete declare statements within procedures are translated into abstract null statements.

Generally, labellists are omitted during translation because the relevant information is contained in the corresponding procedure-declaration-part in the form of an index-list (cf. 6.1.2.1).

(24) trans-p-st(p) =

is-c-p-if-statement∘p(t) → trans-p-if-st(p)

is-c-p-group∘p(t) → trans-p-group(p)

is-c-return-statement∘p(t) → trans-return-st(p)

T → trans-st(p)

for:(is-c-p-if-statement ∨ is-c-p-unconditional-statement)(p(t))

Ref.: trans-st 6-10(34)

(25) trans-p-if-st(p) =

$\mu_0$(<s-st:IF>,<s-expr:trans-expr($s_2 \circ s_1 \circ$p)>,<s-then:trans-p-st($s_2 \circ s_2 \circ$p)>,
<s-else:trans-p-else-st($s_2 \circ s_0 \circ$p)>)

for:is-c-p-if-statement∘p(t)

Ref.: trans-expr 6-14(46)

Note: Balanced statements are a subclass of statements, hence the translation of balanced statements is performed by the function trans-p-st above.

(26) trans-p-else-st(p) =

$\Omega \bullet p(t) \rightarrow \mu_0(\langle s\text{-}st:NULL\rangle)$

$T \rightarrow$ trans-p-st(p)

for: (is-c-p-if-statement $\vee$ is-c-p-unconditional-statement $\vee$ is-$\Omega$)(p(t))

Note: Abstract if-statements possess an else component in any case. The else component is an abstract null statment, if the concrete if-statement has no alternative statement.

(27) trans-p-group(p) =

$is\text{-}c\text{-}p\text{-}simple\text{-}group \bullet p(t) \rightarrow$ trans-p-selist$(s_3 \bullet p)^{\frown}\langle null_0\rangle$

$T \rightarrow$

$\mu_0(\langle s\text{-}iteration:$trans-do-spec$(s_2 \bullet p)\rangle,$
$\langle s\text{-}do\text{-}list:$trans-p-selist$(s_4 \bullet p)^{\frown}\langle null_0\rangle\rangle)$

where:
$null_0 = \mu_0(\langle s\text{-}st \bullet s\text{-}st:NULL\rangle)$

for: $is\text{-}c\text{-}p\text{-}group \bullet p(t)$

Ref.: trans-do-spec 6-12(38)

Note: A simple group within a procedure is translated into a procedure-text-part-list by the function trans-p-selist.

(28) trans-return-st(p) =

$\mu_0(\langle s\text{-}st:RETURN\rangle,\langle s\text{-}expr:$trans-expr$(s_3 \bullet p)\rangle)$

for: $is\text{-}c\text{-}return\text{-}statement \bullet p(t)$

Ref.: trans-expr 6-14(46)

## 6.2 CONSTRUCTION OF THE TEXT-PART-LIST

A text-part-list is a list of text-parts. A text-part has two immediate components, an abstract statement and an abstract text which is a list of character values (cf. chapter 7).

Roughly speaking, the text-part-list of an abstract program is the result of a (nearly one to one) mapping process from the immediate components of a concrete program, i.e., sentences and text (cf. 5-1(2)), into text-parts in the same order of succession, whereby a concrete sentence and the subsequent text up to the next sentence is concentrated into one text-part. The information concentrated in the declaration-part of the program is omitted during this translation process: the labels are omitted, procedures are translated into abstract null statements, and

declare-statements are translated into abstract activate statements, whereby the declarative information is lost.

(29) mk-text-part-list(b) =

$$\text{opt-tpl}_1^{\frown} \overset{n_0}{\underset{n=1}{\text{LIST}}} \text{mk-text-part}(s_n \circ s_3 \circ b)$$

where:
$\text{opt-tpl}_1$ = ($\neg$is-$\Omega \circ s_1 \circ b(t)$ $\rightarrow$ $\langle \mu_0(\langle$s-st$\circ$s-st:NULL$\rangle, \langle$s-text:lin-3$\circ s_1 \circ b(t)\rangle)\rangle$, T $\rightarrow$ $\langle\rangle$)
$n_0$ = slength$\circ s_2 \circ b(t)$

for:is-c-program$\circ b(t)$

Ref.: lin-3 4-4(10)
slength 4-2(4)

Note: Since also the first text-part of the text-part-list must contain an abstract statement, an abstract null statement is added if necessary. This is taken into account by the abbreviation $\text{opt-tpl}_1$.

(30) mk-text-part(p) =

$\mu_0(\langle$s-st:trans-sentence$(s_1 \circ p)\rangle, \langle$s-text:lin-3$\circ s_2 \circ p(t)\rangle)$

for:is-c-sentence$\circ s_1 \circ p(t)$ & is-c-text$\circ s_2 \circ p(t)$

Ref.: lin-3 4-4(10)

Note: The structure of the concrete text, given by selectors of the class elem(i), is linearized by the function lin-3 into a list of character values.

(31) trans-sentence(p) =

is-c-declare-statement$\circ p(t)$ $\rightarrow$ trans-declare-st$\circ s_4 \circ p(t)$

is-c-procedure$\circ p(t)$ $\rightarrow$ $\mu_0(\langle$s-st:NULL$\rangle)$

T $\rightarrow$ trans-st$(s_3 \circ p)$

for:is-c-sentence$\circ p(t)$

### 6.2.1 TRANSLATION OF DECLARE STATEMENTS

Declare statements outside procedures are translated into activate statements with an activation list whose elements have as their rescan component the elementary object T, indicating that "rescan" is desired.

(32) trans-declare-st(dl) =

$$\mu_0(\langle s\text{-}st:ACT\rangle,\langle s\text{-}act\text{-}list: \overset{length(idl_1)}{\underset{i=1}{LIST}} mk\text{-}act(elem(i,idl_1),T)\rangle)$$

where:
$idl_1$ = mk-idl(dl)

(33) mk-idl(dl) =

$$\overset{slength(dl)}{\underset{n=1}{CONC}} (is\text{-}c\text{-}identifier\bullet s_1\bullet s_n(dl) \rightarrow \langle mk\text{-}id\text{-}1\bullet s_1\bullet s_n(dl)\rangle,$$

$$T \rightarrow mk\text{-}id\text{-}1\text{-}list\bullet s_2\bullet s_1\bullet s_n(dl))$$

Ref.: mk-id-1 6-2(4)
mk-id-1-list 6-6(21)

Note: According to the structure of concrete declare statements, given by the predicate is-c-declare-statement, the identifiers contained in the list of concrete declarations dl are collected and formed into a list in the same order of succession.

## 6.2.2 TRANSLATION OF STATEMENTS

(34) trans-st(p) =

is-c-if-statement•p(t) → trans-if-st(p)

is-c-group•p(t) → trans-group(p)

is-c-goto-statement•p(t) → $\mu_0(\langle s\text{-}st:GOTO\rangle,\langle s\text{-}label:mk\text{-}id\text{-}1\bullet s_2\bullet p(t)\rangle)$

is-c-include-statement•p(t) → trans-include-st(p)

is-c-assignment-statement•p(t) → trans-assign-st(p)

is-c-null-statement•p(t) → $\mu_0(\langle s\text{-}st:NULL\rangle)$

is-c-activate-statement•p(t) → trans-act-st(p)

is-c-deactivate-statement•p(t) → trans-deact-st(p)

for:(is-c-if-statement ∨ is-c-unconditional-statement)(p(t))

Ref.: mk-id-1 6-2(4)

### 6.2.2.1 If-statements

(35) trans-if-st(p) =

$\mu_0$(<s-st:IF>,<s-expr:trans-expr($s_2$∘$s_2$∘p)>,<s-then:trans-st($s_3$∘$s_2$∘p)>,
<s-else:trans-else-st($s_3$∘$s_3$∘p)>)

for:is-c-if-statement∘p(t)

Note: The following proposition holds
is-c-balanced-statement ⊃ is-c-statement,
hence the translation of balanced statements is performed by the function trans-st above.

(36) trans-else-st(p) =

is-Ω∘p(t) → $\mu_0$(<s-st:NULL>)

T → trans-st(p)

for:(is-c-if-statement ∨ is-c-unconditional-statement ∨ is-Ω)(p(t))

Note: If the concrete if-statement possesses no alternative statement, this function inserts an abstract null statement.

6.2.2.2 Groups

The third component of a simple group, respective the fourth component of an iterated group, described by the predicate is-c-program, is translated by the function mk-text-part-list, given at the top of section 6.2.

(37) trans-group(p) =

is-c-simple-group∘p(t) → mk-text-part-list($s_3$∘p)^<null-$tp_0$>

T →

$\mu_0$(<s-iteration:trans-do-spec($s_2$∘p)>,
<s-do-list:mk-text-part-list($s_4$∘p)^<null-$tp_0$>>)

where:
null-$tp_0$ = $\mu_0$(<s-st∘s-st:NULL>,<s-text:<>>)

for:is-c-group∘p(t)

Note: null-$tp_0$ corresponds to the concrete end-clause of the group. Its existence is necessary because of possible goto's to the end-clause.

(38) trans-do-spec(p) =

$\mu_0$(<s-contr-var:mk-id-1•$s_1$•p(t)>,<s-init:trans-expr($s_3$•p)>,<s-by:by-expr$_1$>,
<s-to:to-expr$_1$>)

where:
by-expr$_1$ = (is-c-BY•$s_1$•$s_4$•p(t) → trans-expr($s_2$•$s_4$•p),
is-c-BY•$s_1$•$s_3$•$s_4$•p(t) → trans-expr($s_2$•$s_3$•$s_4$•p),
T → Ω)
to-expr$_1$ = (is-c-TO•$s_1$•$s_4$•p(t) → trans-expr($s_2$•$s_4$•p),
is-c-TO•$s_1$•$s_3$•$s_4$•p(t) → trans-expr($s_2$•$s_3$•$s_4$•p),
T → Ω)

for:is-c-do-specification•p(t)

Ref.: mk-id-1 6-2(4)

6.2.2.3 Include statements

(39) trans-include-st(p) =

$\mu_0$(<s-st:INCL>,<s-id-pair-list: $\overset{n_0}{\underset{n=1}{\text{LIST}}}$ trans-lib-spec•$s_n$•$s_2$•p(t)>)

where:
$n_0$ = slength•$s_2$•p(t)

for:is-c-include-statement•p(t)

Ref.: slength 4-2(4)

(40) trans-lib-spec(ls) =

$\mu_0$(<s-id-1:$id_1$>,<s-id-2:$id_2$>)

where:
$id_1$ = (is-c-identifier(ls) → mk-id-1(ls),
¬is-Ω•$s_1$(ls) → mk-id-1•$s_1$(ls),
T → Ω)
$id_2$ = (¬is-Ω•$s_3$(ls) → mk-id-1•$s_3$(ls),
T → Ω)

for:is-c-library-specification(ls)

Ref.: mk-id-1 6-2(4)

6.2.2.4 Assignment statements

(41) trans-assign-st(p) =

$\mu_0$(<s-st:ASSIGN>,<s-lp:mk-id-1∘$s_1$∘p(t)>,<s-rp:trans-expr($s_3$∘p)>)

for:is-c-assignment-statement∘p(t)

Ref.: mk-id-1 6-2(4)

6.2.2.5 Activate statements

Each activation of a concrete activate statement consists of an identifier to be activated, and optional one of the keywords RESCAN or NORESCAN. The result of the translation by the function trans-act is an object possessing an id component and a rescan component which is the truth value F in the case of NORESCAN, else the truth value T.

(42) trans-act-st(p) =

$\mu_0$(<s-st:ACT>,<s-act-list: $\overset{n_0}{\underset{n=1}{\text{LIST}}}$ trans-act∘$s_n$∘$s_2$∘p(t)>)

where:
$n_0$ = slength∘$s_2$∘p(t)

for:is-c-activate-statement∘p(t)

Ref.: slength 4-2(4)

(43) trans-act(act) =

mk-act(mk-id-1∘$s_1$(act),¬is-c-NORESCAN∘$s_2$(act))

for:is-c-activation(act)

Ref.: mk-id-1 6-2(4)

(44) mk-act(id,truth) =

$\mu_0$(<s-id:id>,<s-rescan:truth>)

for:is-id(id) & (is-T ∨ is-F)(truth)

6.2.2.6 Deactivate statements

(45) trans-deact-st(p) =

$\mu_0$(<s-st:DEACT>,<s-id-list:mk-id-1-list∘$s_2$∘p(t)>)

for:is-c-deactivate-statement∘p(t)

cont'd

Ref.: mk-id-1-list 6-6(21)

### 6.2.3 TRANSLATION OF EXPRESSIONS

Because of the structure of abstract expressions, given by the predicate is-expr defined in chapter 7, the precedence rules for the various operators, defined by the productions is-c-expression-one, up to is-c-expression-six, are no more necessary. However, parenthesized expressions still remain because of their special semantics on argument passing.

(46) trans-expr(p) =

is-c-reference•p(t) →

$\mu_0$(<s-id:mk-id-1•$s_1$•p(t)>,<s-arg-list: $\underset{n=1}{\overset{n_0}{\text{LIST}}}$ trans-expr($s_n$•$s_3$•$s_2$•p)>)

is-c-constant•p(t) → trans-const•p(t)

is-LEFT-PAR•$s_1$•p(t) → $\mu_0$(<s-op:trans-expr($s_2$•p)>)

¬is-Ω•$s_3$•p(t) →

$\mu_0$(<s-opr:trans-infix-opr•$s_2$•p(t)>,<s-op-1:trans-expr($s_1$•p)>,
<s-op-2:trans-expr($s_3$•p)>)

T → $\mu_0$(<s-opr:$s_1$•p(t)>,<s-op:trans-expr($s_2$•p)>)

where:
$n_0$ = slength•$s_2$•$s_2$•p(t)

for:is-c-expression•p(t)

Ref.: mk-id-1 6-2(4)
slength 4-2(4)

(47) trans-const(const) =

$$\text{is-c-integer(const)} \rightarrow \sum_{n=1}^{n_0} \text{char-num}\circ\text{elem}(n,\text{const}) \cdot 10 \uparrow (n_0 - n)$$

$$\text{is-c-character-string(const)} \rightarrow \underset{n=1}{\overset{n_1}{\text{LIST}}}\ \text{char}_n$$

is-c-bit-string(const) →

$(n_1 = 0 \rightarrow \text{BIT-NULL-STR},$

$T \rightarrow \underset{n=1}{\overset{n_1}{\text{LIST}}}\ \text{char-bit}(\text{cv}_n))$

where:

$n_0$ = length(const)

$n_1$ = (is-Ω∘elem(2,const) → 0,
T → length∘elem(2,const))

$\text{char}_n$ = ($\text{cv}_n$ = <APOSTR,APOSTR> → APOSTR,
T → $\text{cv}_n$)

$\text{cv}_n$ = (elem(n))∘elem(2,const)

for:is-c-constant(const)

Ref.: char-num 9-26(104)
char-bit 9-26(106)

Note: Constants are translated into values. Values are integer values, or lists of character values, or bit strings. Because of the different semantics of the null elements of character- and bit-data, for the null element of bit-data the elementary object BIT-NULL-STR is introduced.

(48) trans-infix-opr(x) =

is-PLUS(x) → ADD

is-MINUS(x) → SUBTR

is-ASTER(x) → MULT

is-SLASH(x) → DIV

x = <OR,OR> → CAT

x = <GT,EQ> ∨ x = <NOT,LT> → GE

x = <LT,EQ> ∨ x = <NOT,GT> → LE

x = <NOT,EQ> → NE

T → x

## 7. ABSTRACT SYNTAX

The abstract syntax describes the syntactical structure of abstract compile time programs as produced by the function translate defined in the foregoing chapter.

The contents of the abstract syntax is the definition of the predicate

is-program

given by a set of predicate definitions which, applied iteratively, describe the composition of a program by its elementary components.

The elementary components of a program belong to the following classes of elementary objects:

(1) A finite class of elementary objects, denoted by names written in capital letters (e.g., GOTO, MINUS), including the empty list <>.

(2) The infinite class of identifiers, characterized by the predicate is-id.

(3) The infinite class of integer values, characterized by the predicate is-intg-val, and the finite classes of character values and bit values, characterized by the predicates is-char-val and is-bit-val, and the empty bit-string BIT-NULL-STR.

(1) is-program =

(<s-decl-part:is-decl-part>,
<s-text-part-list:is-text-part-list>)

(2) is-decl-part =

{{<id:is-decl> | | is-id(id)}}

(3) is-decl =

is-prop-var ∨ is-entry ∨ is-index-list

(4) is-prop-var =

is-INTG ∨ is-CHAR

(5) is-entry =

(<s-entry-decl:is-ENTRY>) ∨
(<s-body:is-body ∨ is-id>) ∨
(<s-entry-decl:is-ENTRY>,
<s-body:is-body ∨ is-id>)

(6) is-body =

(<s-param-list:is-id-list>,
<s-ret-type:is-INTG ∨ is-CHAR>,
<s-decl-part:is-p-decl-part>,
<s-text-part-list:is-p-text-part-list>)

(7) is-p-decl-part =

({<id:is-p-decl> | | is-id(id)})

(8) is-p-decl =

is-prop-var ∨ is-BUILTIN ∨ is-index-list

(9) is-index =

is-intg-val ∨ is-T ∨ is-F

(10) is-p-text-part =

(<s-st:is-p-st>)

(11) is-p-st =

is-p-group ∨ is-p-text-part-list ∨ is-p-if-st ∨ is-return-st ∨ is-goto-st ∨
is-assign-st ∨ is-null-st

(12) is-p-group =

(<s-iteration:is-iteration>,
<s-do-list:is-p-text-part-list>)

(13) is-iteration =

(<s-contr-var:is-id>,
<s-init:is-expr>,
<s-by:is-expr ∨ is-Ω>,
<s-to:is-expr ∨ is-Ω>)

(14) is-p-if-st =

(<s-st:is-IF>,
<s-expr:is-expr>,
<s-then:is-p-st>,
<s-else:is-p-st>)

(15) is-return-st =

(<s-st:is-RETURN>,
<s-expr:is-expr>)

(16) is-text-part =

(<s-st:is-st>,
<s-text:is-char-val-list>)

(17) is-st =

is-group ∨ is-text-part-list ∨ is-if-st ∨ is-goto-st ∨ is-assign-st ∨
is-null-st ∨ is-act-st ∨ is-deact-st ∨ is-include-st

(18) is-group =

(<s-iteration:is-iteration>,
<s-do-list:is-text-part-list>)

(19) is-if-st =

(<s-st:is-IF>,
<s-expr:is-expr>,
<s-then:is-st>,
<s-else:is-st>)

(20) is-goto-st =

(<s-st:is-GOTO>,
<s-label:is-id>)

(21) is-assign-st =

(<s-st:is-ASSIGN>,
<s-lp:is-id>,
<s-rp:is-expr>)

(22) is-null-st =

(<s-st:is-NULL>)

(23) is-act-st =

(<s-st:is-ACT>,
<s-act-list:is-act-list-1>)

(24) is-act =

(<s-id:is-id>,
<s-rescan:is-T ∨ is-F>)

(25) is-deact-st =

(<s-st:is-DEACT>,
<s-id-list:is-id-list-1>)

(26) is-include-st =

(<s-st:is-INCL>,
<s-id-pair-list:is-id-pair-list-1>)

(27) is-id-pair =

(<s-id-1:is-id ∨ is-Ω>,
<s-id-2:is-id ∨ is-Ω>)

(28) is-expr =

is-infix-expr ∨ is-prefix-expr ∨ is-paren-expr ∨ is-ref ∨ is-value

(29) is-infix-expr =

(<s-opr:is-infix-opr>,
<s-op-1:is-expr>,
<s-op-2:is-expr>)

(30) is-infix-opr =

is-arith-opr ∨ is-comp-opr ∨ is-bit-opr ∨ is-CAT

(31) is-arith-opr =

is-ADD ∨ is-SUBTR ∨ is-MULT ∨ is-DIV

(32) is-comp-opr =

is-GT ∨ is-GE ∨ is-EQ ∨ is-LE ∨ is-LT ∨ is-NE

(33) is-bit-opr =

is-OR ∨ is-AND

(34) is-prefix-expr =

(<s-opr:is-prefix-opr>,
<s-op:is-expr>)

(35) is-prefix-opr =

is-NOT ∨ is-PLUS ∨ is-MINUS

(36) is-paren-expr =

(<s-op:is-expr>)

(37) is-ref =

(<s-id:is-id>,
<s-arg-list:is-expr-list>)

(38) is-value =

is-intg-val ∨ is-char-val-list ∨ is-bit-string

(39) is-char-val =

is-alpham-char ∨ is-BLANK ∨ is-APOSTR ∨ is-EQ ∨ is-PLUS ∨ is-MINUS ∨ is-ASTER ∨
is-SLASH ∨ is-LEFT-PAR ∨ is-RIGHT-PAR ∨ is-COMMA ∨ is-POINT ∨ is-SEMIC ∨
is-COLON ∨ is-AND ∨ is-OR ∨ is-NOT ∨ is-GT ∨ is-LT ∨ is-QUEST ∨ is-PERC ∨
is-extralingual-char

(40) is-alpham-char =

is-letter ∨ is-digit ∨ is-BREAK

(41) is-letter =

is-A-CHAR ∨ is-B-CHAR ∨ is-C-CHAR ∨ is-D-CHAR ∨ is-E-CHAR ∨ is-F-CHAR ∨
is-G-CHAR ∨ is-H-CHAR ∨ is-I-CHAR ∨ is-J-CHAR ∨ is-K-CHAR ∨ is-L-CHAR ∨
is-M-CHAR ∨ is-N-CHAR ∨ is-O-CHAR ∨ is-P-CHAR ∨ is-Q-CHAR ∨ is-R-CHAR ∨
is-S-CHAR ∨ is-T-CHAR ∨ is-U-CHAR ∨ is-V-CHAR ∨ is-W-CHAR ∨ is-X-CHAR ∨
is-Y-CHAR ∨ is-Z-CHAR ∨ is-DOLLAR ∨ is-COMM-AT ∨ is-NUMBER-SIGN

(42) is-digit =

is-0-CHAR ∨ is-1-CHAR ∨ is-2-CHAR ∨ is-3-CHAR ∨ is-4-CHAR ∨ is-5-CHAR ∨
is-6-CHAR ∨ is-7-CHAR ∨ is-8-CHAR ∨ is-9-CHAR

(43) is-bit-string =

is-bit-val-list-1 ∨ is-BIT-NULL-STR

(44) is-bit-val =

is-0-BIT ∨ is-1-BIT

(45) is-extralingual-char =

Note: This predicate is implementation defined. It is equivalent to the predicate is-c-extralingual-character of chapter 5.

# 8. INFORMAL INTRODUCTION TO THE INTERPRETATION OF ABSTRACT COMPILE TIME PROGRAMS

This chapter gives an informal treatment of the concepts used in the formal definition of the interpreter in chapter 9. The informal treatment is separated from the formal treatment to allow a compact formal part and also to explain concepts which are not concentrated in a single part of the formal definition but influence the mechanism as a whole.

Also the syntax of abstract compile time programs given in chapter 7 is illustrated and the most important relations between concrete and abstract syntax are explained in order to give the reader who is familiar with the concrete syntax an intuitive idea of the interpretation process, without presupposing knowledge about the translator.

It is not the aim of the informal discussion to define compile time facilities completely. Most detailed questions can be answered by studying the formal defintion of the interpreter.

## 8.1 STRUCTURE OF ABSTRACT COMPILE TIME PROGRAMS

An abstract compile time program, in the following called program, is an object described by the predicate is-program given in chapter 7. It consists of two immediate components, the declaration-part and the text-part-list.

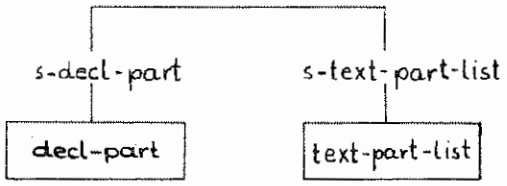



Fig. 1 Main structure of a program

### 8.1.1 THE TEXT-PART-LIST

The text-part-list is a list of text-parts. A text-part consists of a statement and of a text.

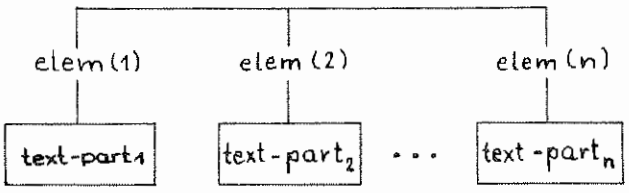



Fig. 2 Structure of a text-part-list

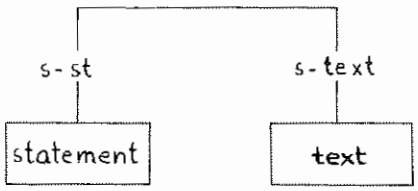



Fig. 3 Structure of a text-part

A text-part corresponds in a concrete program to a sentence (cf. section 3.2.1) and the subsequent text up to the next sentence. If a sentence immediately is succeeded by a further sentence in a concrete program, then the text is the empty list <>. If the concrete program does not begin with a sentence but rather with a concrete text, then the statement of the first text-part is a null statement.

A text is only a list of character values. All necessary grouping, e.g., identifiers, argument lists, comments, strings will be done dynamically by the interpreter (cf. section 9.9).

Throughout the formal definition the term statement denotes a logically complete unit of a program to be executed during the sequential flow of control at the point given by its position within the program. The term includes: The simple statement (e.g., assignment statement, goto statement, null statement), the if-statement, and the different types of do-groups. So, the term "statement" does not denote the units syntactically delimited by semicolons in the concrete program, but logical units that may appear anywhere "in a statement context", e.g., as THEN alternative of an if-statement, and that may in some way be executed independently from other program parts.

Statements may themselves contain statements (namely the if-statement) or even text-part-lists (namely the group). Since these contained statements principally may be any type of statements and thus may themselves contain statements or text-part-lists, the text-part-list of a program may be not just a linear sequence of text-parts but a rather complex structure of nested statements and text-parts.

The different types of statements are not described in detail here because in most cases they are the result of a nearly one-to-one mapping of the corresponding concrete statements. The following are only some additional remarks, mentioning some deviations between the abstract and concrete syntax.

Group and text-part-list. There are two essentially different "do-groups" in a concrete program: Those with iteration specification and those without it. Only those with iteration specification are translated into groups. Those without iteration specification are translated into text-part-lists. Thus a statement may itself be just a text-part-list in the abstract program.

If-statement. The if-tstatement has always two alternative statements. If there is no else alternative specified in the concrete program, the translator inserts a null statement.

Null statement. A null statement in an abstract program may result not only from a concrete null statement, but also from a missing else alternative of an if-statement, from a concrete declare statement inside procedures or from an end clause. Declare statements and end clauses may be labelled in a concrete program and hence may not be simply omitted on translation. Furthermore, also procedures are replaced by null statements on constructing the text-part-list (procedure bodies constitute declarations and hence belong to the declaration-part of the program).

Activate statement. Also an activate statement in an abstract program may result not only from a concrete activate statement, but also from a concrete declare statement outside procedures. Declare statements outside procedures play a double role. On the one hand they declare identifiers to denote a variable or

entry name, on the other hand they activate these identifiers and indicate RESCAN each time the flow of control reaches the declare statement. On translation the declarative information is collected in the declaration-part and the "dynamic" information which remains is reflected in an activate statement whose single activations have the truth value T as their rescan component.

### 8.1.2 THE DECLARATION-PART

In the declaration-part of a program all declarative information is collected, including the bodies of the procedures which themselves possess declaration-parts containing the declarative information which is local to the procedures. Each identifier declared outside procedures, whether its declaration in the concrete program is explicit or implicit, has a declaration in the declaration-part of the program.

To each identifier of the concrete program corresponds uniquely an abstract identifier which is an elementary object satisfying the predicate is-id. The transformation from the concrete identifier to its corresponding abstract identifier is performed by the function mk-id, given in section 2.2. In the following the term identifier denotes such an abstract identifier, while the identifiers of the concrete program are denoted as concrete identifiers where necessary.

The structure of a declaration-part is the following: Each declared identifier serves as selector selecting its declaration from the declaration-part.

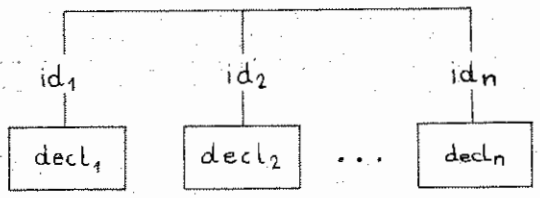



Fig. 4 Structure of a declaration-part

This structure of a declaration-part provides easy access to an individual declaration through the declared identifier itself; any other structure would require a more complicated device for accessing an individual declaration.

Each individual declaration is an object, whose structure depends essentially on the type of the declaration. There are the following types of declarations:

Proper variables, in the following called variables,
entry names,
statement labels, in the following called labels.

#### 8.1.2.1 Variables

The declaration of a variable is simply one of the both elementary objects: INTG and CHAR. The translator gets the information from the corresponding concrete declare statement. In the case of INTG the variable is predestinated to hold as its value an integer value (only decimal integer arithmetic of precision (P,0) is performed in the compile time facilities), in the case of CHAR a char-val-list (only varying character strings that have no maximum length are possible).

#### 8.1.2.2 Labels

The declaration of a label is an index list (list of positive integers and truth values) which localizes that statement of the text-part-list of the program which was labelled in the concrete program. The index list is constructed by the translator from the position of the labelled statement within the concrete program.

8.1.2.3 Entry names

The declaration of an entry name is an object with (in general) two immediate components.

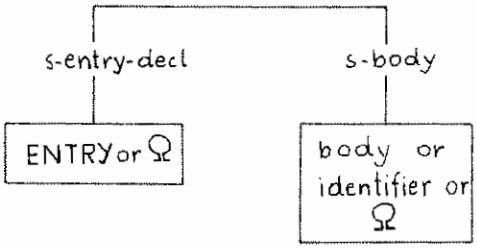



Fig. 5 Declaration of an entry name

At most one of the both immediate components may be Ω, i.e., does not exist:

The existence of the entry-decl component indicates that the concrete program contains a declare statement defining the corresponding concrete identifier to denote an entry name, i.e., it is associated with the attribute ENTRY within the declare statement.

A missing body component indicates that no procedure is specified within the concrete program whose entry name list contains the entry name specified by the declare statement. The missing body may appear within the declaration-part of an external program which may be incorporated during the computation by means of an include statement.

The case that the body component is solely an identifier may appear if the corresponding concrete procedure specifies at least two entry names. One entry name, namely the first one in the entry name list, is associated with the translated procedure, i.e., with the body, in the declaration-part, all other entry names get this special entry name as their body component instead of the body itself. This indirect step from an entry name via another entry name to the corresponding body is of advantage for the interpreter, because any body gets only one entry in the state component "procedure body directory", independent of whether the body is associated with more than one entry name or not.

The structure of a procedure body is discussed in the following.

8.1.2.4 Procedure bodies

A procedure body, in the following called body, corresponds in a concrete program to a procedure (without regard to the entry name list). A body consists of four immediate components:

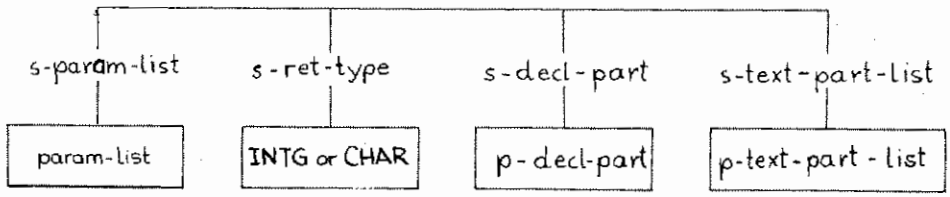



Fig. 6 Structure of a body

(1) The parameter list. It is the list of those parameter identifiers to whom arguments are passed when the body is activated by means of a reference to the body.

(2) The return type. It specifies the type of value to which the function value is to be converted on return to the point of invocation.

(3) The procedure-declaration-part, shortly called p-declaration-part. It contains all declarations local to the body. A p-declaration-part differs from a declaration-part (of a program) in that, first, a p-declaration-part may not contain declarations of entry names, and second, may contain (in contrast to a declaration-part) builtin declarations, i.e., the following types of declarations may occur:

Variables,
labels,
builtin function names.

The declaration of a builtin function name is solely the elementary object BUILTIN.

(4) The procedure-text-part-list, shortly called p-text-part-list. A p-text-part-list is a list of p-text-parts. A p-text-part has only one immediate component, namely a p-statement.

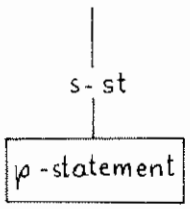



Fig. 7 Structure of a p-text-part

A p-text-part corresponds in a concrete procedure to a p-sentence. No text to be scanned by the replacement mechanism exists within procedures. The similar strncture of text-parts and p-text-parts, by using in both cases the same selector s-st, is of advantage for the interpreter.

The class of p-statements differs from the class of statements in the following points:

(a) The activate, deactivate, and include statements do not belong to the class of p-statements, i.e., they may not occur within a p-text-part-list.

(b) The return statement belongs only to the p-statements.

(c) The p-if-statement and the p-group may not contain statements and text-part-lists, but rather p-statements and p-text-part-lists.

### 8.1.3 EXPRESSIONS

Concrete expressions are decomposed by the translator into (possibly nested) "elementary expressions". There are five different forms of elementary expressions:

(1) An infix expression, consisting of two operand expressions and an infix operator which is an elementary object.

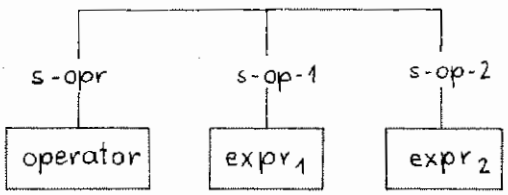



Fig. 8 Infix expression

(2) A prefix expression, consisting of an operand expression and a prefix operator which is an elementary object.

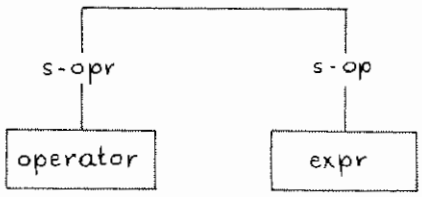



Fig. 9 Prefix expression

(3) A parenthesized expression, consisting of an operand expression only.

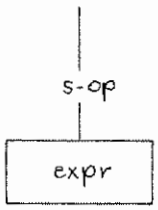



Fig. 10 Parenthesized expression

In principle, the parentheses of a concrete program could be eliminated by the translator producing structured objects as already described. But since in the language there is one case (argument passing) where parentheses have more than syntactical meaning, the parenthesized expressions are left in the abstract program in the form of an object having only one component, namely the translation of the concrete expression contained in the parentheses.

(4) A reference, consisting of two immediate components, the identifier and the argument list which is a list of expressions (the empty list <> included).

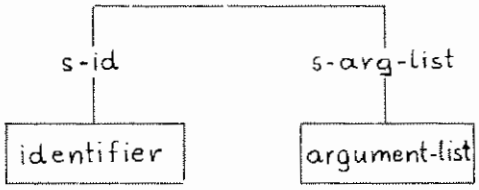



Fig. 11 Reference

A reference may refer to a variable (the argument list is the empty list <>), to a procedure, or to a builtin function.

(5) A value, corresponding in a concrete program to a constant. There are three types of values

(a) integer values
(b) char-val-lists
(c) bit-strings

These three classes of values are mutually disjoint, i.e., no further "data attributes" are necessary. To achieve this the null bit string is elementary object BIT-NULL-ST, rather than the empty list <>.

## 8.2 DYNAMIC PROPERTIES OF IDENTIFIERS AND THEIR INFLUENCE ON THE STATE

### 8.2.1 SCOPE OF IDENTIFIERS

The scope of an identifier declared in the main program is the entire text-part-list of that program and the p-text-part-lists of those bodies which do not redeclare that identifier. If there are external programs to be included by means of include statements, then their text-part-lists and the p-text-part-lists of those bodies which are contained in the external programs and do not redeclare that identifier belong to the scope of that identifier too.

The scope of an identifier declared in a body is limited to the corresponding p-text-part-list.

The scope of an identifier declared in an external program is the same as though this identifier were declared in the main program. A redeclaration of an identifier by means of an external program is not legal and would lead during the interpretation of that external program to a multiple declarations error. Note however, that if an identifier is declared within an external program, this declaration is not "known" until the first interpretation of the corresponding declaration-part.

An identifier that is declared in a declaration-part will in the following be called global. If it is declared in a p-declaration-part, then local.

### 8.2.2 DENOTATION OF IDENTIFIERS

The denotation of an identifier represents the entire information associated with the identifier except of its scope. In general the denotation contains static information as well as dynamic information. Static information remains unchanged during the entire interpretation process. The information contained in the declaration of an identifier is static. Dynamic information, as e.g., the value of a variable, may be altered during the interpretation process. In the following the five different types of denotations are discussed.

(1) Denotation of a variable:

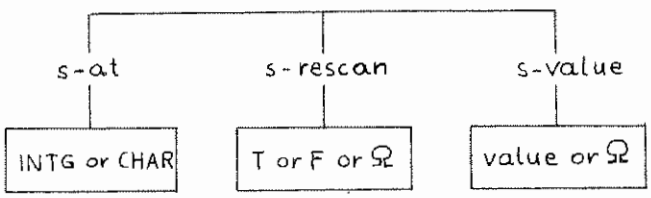



Fig. 12 Denotation of a variable

The at component represents the type of the variable specified by the corresponding declaration. It is the only static component.

When a variable (necessarily global), an entry name, or a builtin function name is encountered during the scan of a text, the question whether it is activated or deactivated is of importance. In the first case the substitution process becomes active in which case a further distinction is made, namely whether the value of the reference must be scanned for replacements or not. This information is reflected by the rescan component: Ω denotes "not activated", T denotes "activated and scan of the replacement value", and F denotes "activated and no scan of the replacement value". The rescan component may be altered by activate and deactivate statements. Initially, i.e., immediately after the interpretation of the declaration-part, this component is Ω. In the case of a local variable it remains so.

The value component presents the value of the variable. Initially the value component is Ω. It may be altered by an assignment to the variable.

(2) Denotation of an entry name:

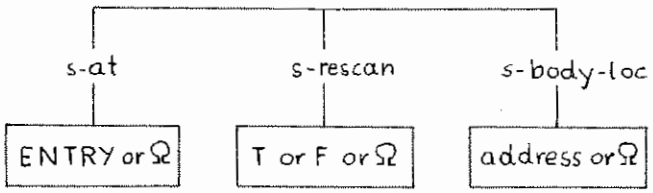



Fig. 13 Denotation of an entry name

The at component indicates whether the entry name is declared in the concrete program by means of a declare statement. The body-loc component presents the address under which the corresponding body is stored in the "procedure body directory" P. This component is Ω if no body exists; in this case the at component is ENTRY. The at and the body-loc components are static only with regard to one program execution. On incorporating a new external program these components could be altered if they were Ω.

(3) Denotation of a builtin function name:

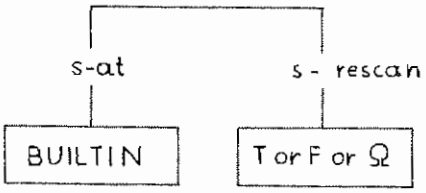



Fig. 14 Denotation of a builtin function name

Only an identifier which is the abstract representation of one of the concrete identifiers INDEX, LENGTH, BUILTIN can be associated with such a denotation. This denotation is created either on the interpretation of a p-declaration-part from the declaration of a builtin function name, or on the interpretation of an activate or deactivate statement, or on the evaluation of a reference appearing in an expression. The at component is static.

(4) Denotation of a label:

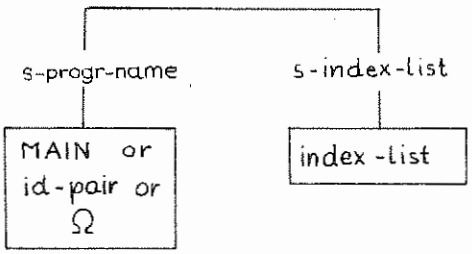



Fig. 15 Denotation of a label

The entire denotation is static. The progr-name component presents the name of the program in which the label is declared: MAIN in the case of the main program, or a pair of identifiers specified by an include statement in the case of an external program. For local labels this component is Ω. The index-list component presents the index-list taken from the corresponding label declaration. It represents the location of a (p-)statement relative to the text-part-list of a program or to the p-text-part-list of a body.

(5) Denotation of a dummy:

The entire denotation is solely a value, either of the type INTG (is-prop-intg-val) or of the type CHAR (is-char-val-list). A dummy is used either for saving the function value when the interpretation of the body is finished and the dump is popped up, or for storing intermediate results during the application of the scan and replacement mechanism to the arguments of a reference within a text.

### 8.2.3 THE ENVIRONMENT AND THE DENOTATION DIRECTORY

The association of an identifier with a denotation, initialized during the interpretation of the declatation-part of a program, holds during the interpretation of the whole text-part-list of the program, unless a procedure body becomes active whose p-declaration-part redeclares that identifier. In this case the old (global) denotation must be saved and the new one is initialized according to the p-declaration-part. Because of the scope rules (see 8.2.1) the old denotation of the identifier is reestablished when the control leaves the procedure.

In the interpreter this situation is taken into account by means of the two state components "environment" E and "denotation directory" DN.

The environment associates all identifiers which have a denotation with addresses. This mapping is realized in the same way as in the case of a declaration-part associating identifiers with declarations.

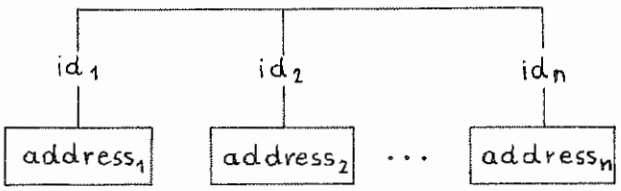



Fig. 16 The environment

The denotation directory associates addresses with denotations.

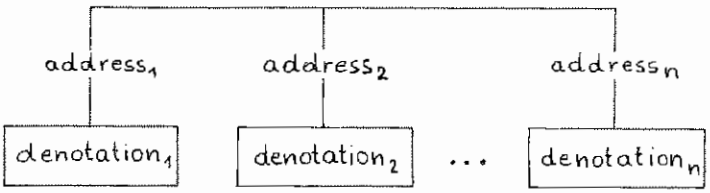



Fig. 17 The denotation directory

This indirect step

$$id \xrightarrow{E} ad \xrightarrow{DN} dn$$

from an identifier via an address to the denotation, together with the fact that an environment can be dumped (within another state component called "dump") and that a modified environment can be established, fulfills the above requirements.

During the interpretation of the declaration-part of a program each identifier declared there is associated at first with a proper address in the environment. Under this address the denotation of that identifier is initialized in the denotation directory. When during the following interpretation of the text-part-list a certain component of the denotation of an identifier is required or a dynamic component of the denotation has to be modified, the denotation is available by applying the identifier (using it as a selector) to the environment and again applying the result, an address, (also used as a selector) to the denotation directory.

The necessary addresses are the result of a one-to-one mapping (cf. section 2.2) from a pair consisting of the identifier for which the address is required and the scope information (global or the address of a body). One hidden feature should be noted at this point: Entries of the DN component of the state are never erased. The use of this one-to-one mapping ensures the "static storage class" for variables. (Every time a function call of one and the same procedure is to be interpreted, which in consequence leads to the interpretation of the p-declaration-part, a new updating of E is performed. But the mapping for a local identifier always specifies the same address. Therefore, a reference to DN always gives the same entry which had been established by the first execution of that procedure).

When control is transferred into a body by means of a function reference the environment is copied before it is stored in the dump. The copy which is modified according to the p-declaration-part of that body is established as the new environment. More exactly, each identifier declared in the p-declaration-part causes a modification of the copy in one of the two following ways:

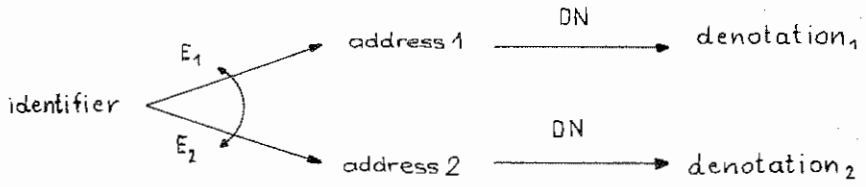
If the identifier was not known till now, i.e., had no entry in the old environment, then the copy is enlarged by the identifier and its associated address.

If the identifier was known, i.e., it is redeclared by the p-declaration-part, then the new address is substituted in place of the corresponding old address into the copy.

In both cases, the corresponding denotation of the identifier is initialized according to the p-declaration-part and is stored under the new address in the denotation directory.

When the p-text-part-list is interpreted and control reaches the text-part-list of the program again, the local environment is replaced by the dumped environment and the interpretation of the text-part-list is continued.

The necessary "change of environments" in the case of a procedure entry or procedure exit, acts for an identifier redeclared within the procedure like a switch which associates that identifier in both positions with an address each:




Both addresses are connected in the denotation directory with a denotation each, which are different in general.

## 8.3 INTERPRETATION OF THE DECLARATION-PART

The interpretation of the program, which is contained (together with possible external programs) in the initial state $\xi_0$, starts with the interpretation of the corresponding declaration-part.

Each identifier declared in the declaration-part is associated with an address by means of the function af (cf. section 2.2). Each such identifier together with its address constitutes an entry into the environment. Any declaration which is associated with an identifier in the declaration-part becomes a part of the denotation which is initialized now and is stored under the corresponding address in the denotation directory. In case the declaration contains a body an entry is also made in the procedure body directory under the same address. The execution status of this entry is initialized with F.

The rescan component (in the case of variables, entry names, and builtin function names), as well as the value component (in the case of variables) is not initialized, i.e., is $\Omega$.

In the case of a label the name of the program is assigned to the progr-name component of the corresponding denotation.

The interpretation of the declaration-part is finished after the above procedure is done for all components of the declaration-part. Since the whole information the declaration-part represents is stored in the three state components E, DN, and P, the declaration-part is of no further use and therefore is erased.

## 8.4 INTERPRETATION OF THE TEXT-PART-LIST

As described in section 8.1.1 the text-part-list of a program constitutes a rather complex system of nested statements and text-parts, since some types of statements may themselves contain statements of any type or even text-part-lists. The present chapter describes the flow of control of the compile time machine through this system of statements and text-parts.

The normal flow of control is influenced by:

(1) the sequencing of text-parts within a text-part-list,

(2) the nesting of text-part-lists within statements,

(3) the nesting of statements occurring as then and else alternatives in if-statements,

(4) the iteration specification of groups,

(5) the incorporation of external programs by include statements.

The flow of control is governed by a state component, the control information CI. It reflects the current status of the compile time machine with respect to these five points.

Additionally, the flow of control may be modified abnormally by means of the goto statement. This is performed essentially by modifying the control information CI.

### 8.4.1 SEQUENTIAL INTERPRETATION OF TEXT-PARTS

The sequential interpretation of text-parts within a text-part-list is governed by two components of the control information CI, the txt component and the index component. Whenever a text-part out of a text-part-list (but not a nested statement or text-part contained within it) is under execution the txt component is that text-part-list and the index component is the number of the currently interpreted text-part within the list; e.g., when the third text-part of the text-part-list is executed the txt component is the text-part-list and the index component is the integer value 3.

During the execution of a text-part-list the txt component of CI is in general left unchanged, while the index component is always updated between two text-part executions.

Whenever the execution of a text-part (except the last one of the text-part-list) has been terminated, the index component is increased by 1, the text-part denoted by the new index now is executed. The execution of a text-part consists of:

(1) execution of the statement contained in the text-part,

(2) application of the scan and replacement mechanism to the text contained in the text-part, if sequential flow of control is supposed.

### 8.4.2 NESTING OF TEXT-PART-LISTS

When a text-part-list is to be executed it is entered into the txt component of CI, the index component of CI is initialized to 1 and the mechanism described above is started.

These actions are not sufficient in the case where a text-part-list is to be executed during the execution of a statement which itself is contained within a text-part-list. Because in this case the txt and index components of CI keep the information needed for the sequential execution of the text-parts of the containing text-part-list. This information would be lost by overwriting, if no special provisions where made when the nested text-part-list is executed. In order to keep the txt and index for the containing text-part-list and also information in the control specifying how to continue after termination of the nested text-part-list, the control information CI is handled as a stack: Whenever the execution of a text-part-list starts, before the txt and index components of CI are overwritten, the complete current control information CI and control C are copied into two additional components of CI. When the last text-part of the nested text-part-list has been executed, these two components are reinstalled as state components CI and C and the execution of the containing text-part-list continues correctly.

Thus, the control information (apart from one special component, the progr-name component, which is used only for goto statements outside procedures) consists of four components: The current txt and index, and the control information and control of the containing text-part-list. Again this control information of the containing text-part-list consists of four such components, and so forth. Each level in the control information represents one level in the system of nested text-part-lists.

### 8.4.3 THE IF-STATEMENT

The if-statement introduces into the language a form of statement nesting differing from the nesting of text-part-lists. In order to reflect this form of statement nesting too, the concepts of the txt and index components of CI are slightly modified: The txt component may not only be a text-part-list but also an if-statement. In this case, the index component is not an integer, but rather a truth value, the index T denoting the then component and the index F denoting the else component.

The execution of an if-statement causes the following actions to be performed:

(1) The decision expression of the if-statement is evaluated and converted to a truth value.

(2) The same actions, i.e., pushing down the control information for one level, are performed as described for the execution of a text-part-list in the foregoing subsections with the following changes:

  (a) The if-statement (instead of a text-part-list) is entered into the txt component of CI;

  (b) the index component is initialized to the truth value computed in (1);

  (c) the meaning of "index denotes a text-part out of the txt component" is extended as explained above;

  (d) both the then and else components are considered as "last" statements of the txt component, i.e., the control information CI is popped up after termination of either of them.

### 8.4.4 STRUCTURE OF THE CONTROL INFORMATION CI

The control information CI consists of five immediate components: txt, index, control information, control, and program name, where the contained control information again consists of these five components, and so forth. Each contained level in the control information represents a containing level in the system of nested statements and text-parts. It ends up with the level representing the outermost text-part-list of the main program (or inside procedures the outermost p-text-part-list of the body).

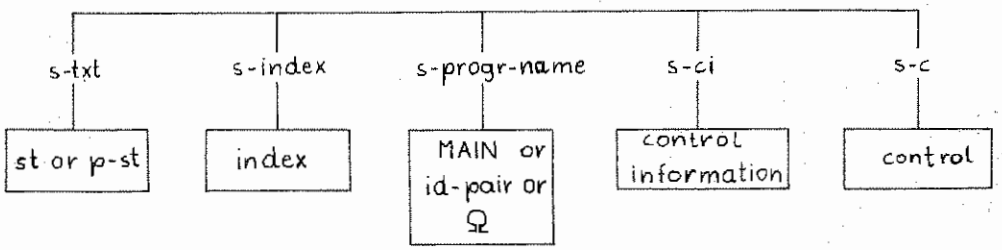



Fig. 18 Structure of the control information

The prog-name component is different from Ω only if the txt component represents the outermost text-part-list of the main program or of an external program. In this case the progr-name component presents the name of the program, i.e., MAIN in the case of the main program, or a pair of identifiers specified by the corresponding include statement in the case of an external program. A local control information, i.e., inside procedures, does not possess this component. This component is necessary only for the interpretation of goto statements.

Either the txt component is a (p-)text-part-list and the index component is an integer, or the txt component is a (p-)if-statement and the index component is a

truth value. The index component denotes that statement or text-part out of the txt component which is currently under execution.

Example:

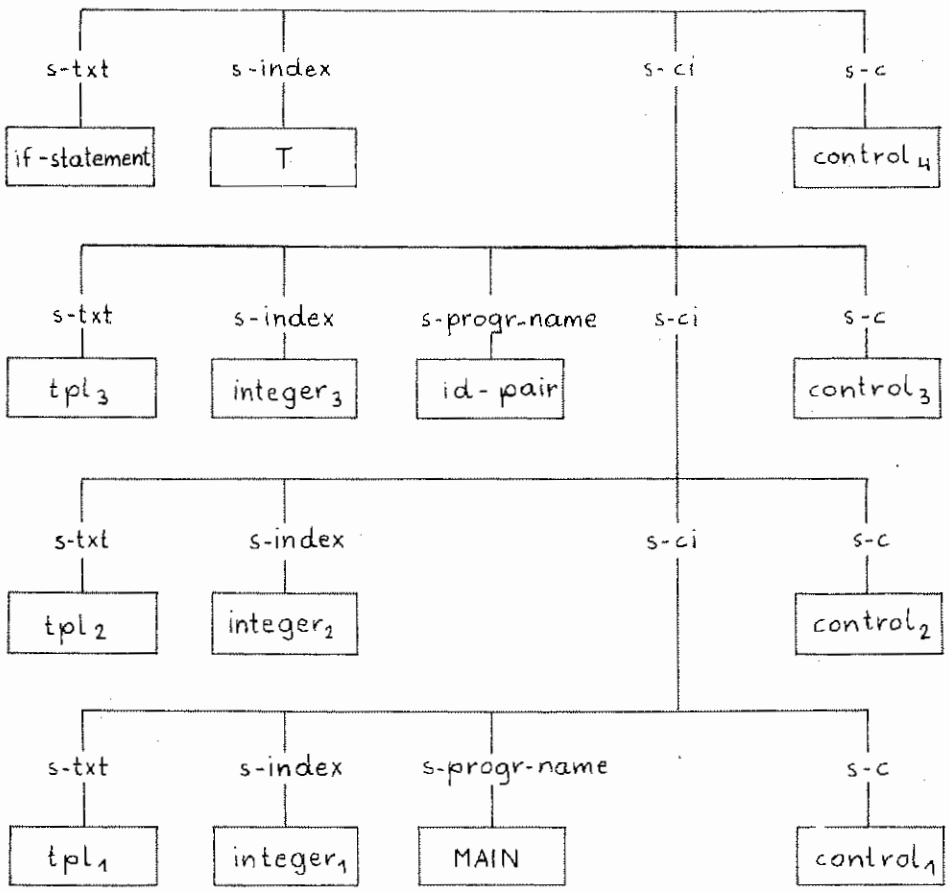



Fig. 19 Example of a global control information

This example presents a global control information, i.e., a control information that may occur outside procedures. $tpl_1$ is the outermost text-part-list, i.e., the text-part-list of the main program, $integer_1$ is an integer denoting that text-part of $tpl_1$ which is currently under execution. The st component of this text-part is either a text-part-list or a group containing a text-part-list, this text-part-list is denoted by $tpl_2$. The text-part to which the integer $integer_2$ points within $tpl_2$ contains as its st component an include statement which specifies the pair of identifiers id-pair identifying an external program. This external program had been incorporated and its text-part-list $tpl_3$ constitutes the next level of CI. This level possesses a progr-name component indicating, that this and any further levels without such a component belong to an external program identified by id-pair. The integer $integer_3$ points to a text-part within $tpl_3$ whose st component is an if-statement whose then component is interpreted.

So the control information CI denotes exactly the innermost text-part or statement (in the case of an if-statement) currently being executed. Since the txt components of levels without a progr-name component are already uniquely determined by the text-part-list of the program to which they belong and by the

index components, those txt components are redundant. They are always copied for convenience of use.

The way of localizing a text-part or statement relative to the text-part-list of the main program or of an external program by a list of indices is used in the declaration of labels and in the execution of the goto statement.

### 8.4.5 THE GOTO STATEMENT

A goto statement consists essentially of an identifier whose denotation is that of a label (in proper cases).

The denotation of a local label, i.e., local to a body, has one immediate component: An index list giving the statement location relative to the p-text-part-list of the body.

The denotation of a global label has two immediate components:

(1) The name of the program in which the label was declared.

(2) The index list giving the statement location relative to the text-part-list of the program identified by the program name.

The aim of a goto statement is to simulate the normal flow of control to the target statement denoted by the label denotation, i.e., to transform the compile time machine into that state in which it would have been if the target statement would have been encountered normally. This means first to look for that text-part-list within CI which belongs to the program identified by the program name of the label denotation, and second, to bring the control information into such a form, that the sequence of its indices, if starting from the level marked by the program name and going up to the top, is the index list of the label denotation. This is performed in the following steps:

(1) The levels of text-part-lists and if-statements reflected in the control information CI are terminated one by one until a level is reached which belongs to the program identified by the program name of the label denotation. If the control information is exhausted without success, the program is in error. This occurs if the goto statement refers to an external program which is not currently under interpretation.

This point has to be performed only on the interpretation of a goto statement outside procedures.

(2) Again, levels of CI are terminated one after the other, until the target statement is contained (possibly nested) within the innermost not yet terminated text-part-list or if-statement. This is performed by popping up the control information CI level by level, until the sequence of indices contained in CI and belonging to the program into which the goto shall lead (except the current index) is equal to an initial portion of the index list of the label denotation. (This is the case at latest when the current CI offers in its progr-name component the program name of the label denotation.)

(3) The text-part-lists and if-statements containing the target statement are entered level by level until the innermost of them is reached. This is for each level performed by:

(a) changing the current index component of CI to the value given by the corresponding place in the index list of the label denotation,

(b) stacking the control information CI for one level and entering into the txt component of CI the statement out of the old txt component which is denoted by the just changed index. This statement has to be a (p-)text-part-list or (p-)if-statement if the program is not in error. In particular it cannot be a (p-)group (a goto into a group is forbidden).

(4) Finally, the current index is adjusted, i.e., set to the last value of the index list of the label denotation, and the normal flow of control is continued.

It should be mentioned that, of course, in special cases one or more of these steps may be skipped. In particular, in the simplest case of a goto, namely a goto within the current (p-)text-part-list, only step 4 is applicable.

Note, that a goto statement must not lead out of a procedure. This would be the case if the label denotation had a program name.

### 8.4.6 THE GROUP

A group is a statement specifying repeated execution of a (p-)text-part-list. The two immediate components of a group are: The (p-)text-part-list to be iterated and the iteration specification. After each execution of the (p-)text-part-list the value of a given variable, the controlling variable, is incremented by a given value. The (p-)text-part-list is executed repeatedly until the value of the controlling variable exceeds a given value.

The execution of a group is performed in that way that all actions controlling the iteration of the (p-)text-part-list are performed at the level of the control information CI which is the current one at the point when the execution of the group starts. Each time when the iterated (p-)text-part-list is to be executed, the control information is stacked for one level, i.e., the (p-)text-part-list is executed exactly as described in 8.4.2. During the execution of the iterated (p-)text-part-list, the control component of CI specifies the actions controlling the iteration of the (p-)text-part-list, in particular, it contains the information about the current status of the iteration. Each time when the execution of the iterated (p-)text-part-list terminates, the control information is popped up for one level as described in 8.4.2. Thereby at the popped up level the iteration control is performed.

### 8.4.7 THE INCLUDE STATEMENT

An include statement specifies a list of identifier-pairs. Each pair corresponds to an external program in the state component external program directory EP.

The interpretation of an include statement comprehends, in a successive order beginning with the first identifier-pair, the mapping of any pair onto a selector, which applied to EP yields the associated external program that has to be interpreted.

When a program is incorporated, a new level of the control information CI is established, i.e., the control information CI and control C are stacked into CI, and the text-part-list of the program is entered as the txt component of CI. Furthermore, the program name, i.e., the identifier-pair specified by the corresponding include statement, is entered as the progr-name component of CI. By that, on interpreting a goto statement, the outermost text-part-list of that program can be found in which the label was declared.

Before the interpretation of the program may start with the interpretation of its declaration-part a check must be made whether all components of the declaration-part are compatible with the three state components E, DN and P which together contain the information of all declaration-parts interpreted up to now. Because of the global scope of identifiers declared outside procedures together with the fact that multiple declarations are fobidden, for each identifier declared in the declaration-part of the included program one of the following three conditions must hold:

(1) The identifier has no entry in the environment, i.e., it was never declared in a declaration-part till now.

(2) The identifier has an entry in E and its denotation stored in DN represents an entry declaration, and the corresponding component of the declaration-part under consideration represents a body or an address to the body. (That is exactly the case, when an entry name was declared former and now the corresponding procedure body follows.)

(3) The identifier has an entry in E and its denotation specifies only a body, and the corresponding component of the declaration-part under consideration represents an entry declaration. (This is the case, when a procedure body was specified previously and now the corresponding entry declaration follows.)

Under the assumption that the conditions above are fulfilled, the declaration-part is interpreted. The interpretation begins with an update of the environment for all those identifiers declared in the declaration-part and not occurring in E up to now. Then the entries in the denotation directory and possibly in the procedure body directory are made for each identifier which is associated with a declaration in the declaration-part, as described in section 8.3.

The interpretation of the text-part-list is done in the usual way. It is terminated after the last text-part is interpreted, unless a previous goto statement transfers control out of the included program. In the first case the "next" external program - if there is one - is incorporated and interpreted as described above.

The recursive use of an include statement is allowed, i.e., in the text-part-list of a program incorporated by a certain include statement may occur an include statement specifying the same program. Note that the recursive use of an external program is legal only if the corresponding declaration-part is the null object, i.e., is Ω. Otherwise, the multiple declarations check described above leads to an error.

## 8.5 REFERENCE TO FUNCTIONS

### 8.5.1 REFERENCE TO A PROCEDURE OCCURRING IN AN EXPRESSION

A procedure reference consists of an entry name, i.e., an identifier which currently is associated with the denotation of an entry name (cf. section 8.2.2), and a list of expressions specifying the arguments to be passed to the parameters of the called procedure body.

A procedure reference is proper if:

(1) The body location exists within the entry denotation, and under this body location a body is stored in the procedure body directory P. (The body-loc component of the denotation could be missing, because of the possibility of the separate declaration of entries and their associated body within different declaration-parts.)

(2) The execution status of the body, i.e., the execution component of the corresponding entry of the procedure body directory, is F, indicating that the body is currently not under interpretation. In this way any recursive invocations of a body are detected.

(3) The number of arguments is equal to the number of parameters specified by the param-list component of the body.

(4) The body itself is proper, i.e., all parameters are mutually different and denote variables (in the p-declaration-part of the body) and if the p-declaration-part of the body contains a builtin declaration then the associated identifier must be the abstract representation of one of the concrete identifiers: INDEX, LENGTH, SUBSTR.

If the reference is proper, its arguments are evaluated from left to right. For the evaluation of an argument the declaration of the corresponding parameter

within the p-declaration-part of the body is necessary. Two distinct cases may occur:

(1) The argument expression is a reference to a variable and its attribute, i.e., either INTG or CHAR found in the at component of the denotation of the variable, matches the declaration of the corresponding parameter. In this case the address of the variable is passed to the corresponding parameter. Any use of those parameters will result in a reference to the denotation of the argument. (Passing of address)

(2) In all other cases the argument expression is evaluated and converted to the type given by the declaration of the corresponding parameter. (Passing of value)

The resulting list of addresses and values now is passed to the body. Before the interpretation of the body starts, a unique name, to be used as address of a dummy entry in DN, is created and passed to the called body. After return from the called body the return value of the body is available via this unique name in the denotation directory.

## 8.5.2 INTERPRETATION OF THE PROCEDURE BODY

### 8.5.2.1 The dump D

During the interpretation of a body some components of the current state are used to hold the necessary local information which is of no further use when the interpretation of the body is finished. When this is the case, the former contents of these state components must be reinstalled in order to secure proper continuation of the interpretation. The storage necessary for the state components to be reinstalled when the execution of the body is terminated, is called dump D.

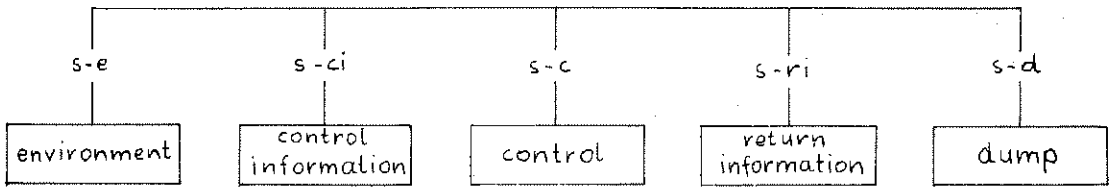



Fig. 20 Structure of the dump

If the reference to a procedure occurs outside procedures, then neither a dump nor a return information exists in the current state. Hence, also the dump to be constructed on entering the body does not contain these components. If the reference to a procedure occurs during the interpretation of a procedure body, then also the current dump and the current return information must be saved when entering the new body.

The dump is an object manipulated as a push-down stack; it maintains dynamically the history of the still active body activations. Its dump component has the same structure and consists of the local state components of the predecessor of that body activation. The state components saved at the bottom of the dump are the environment, the control informtion, and the control to be used outside procedures.

When a body activation is terminated, the components of the dump are copied into the corresponding state components of the compile time machine. All parts of the dump are thus popped up one level.

8.5.2.2 Installation of the body

As mentioned above, the five current state components environment, control information, control, return information, and dump are saved in the dump. Then these state components are initialized as follows:

E must be initialized with the environment current at the time the procedure was declared, because any identifier not declared inside the procedure will have the denotation specified via that environment. The proper environment is always the outermost one, because procedure declarations may occur only in a declaration-part, and never inside a p-declaration-part. In the case that the function reference under consideration occurred in a text-part-list (D is Ω), the proper environment is the current one, but if the reference occurred in a p-text-part-list (D is not Ω) the proper environment is that one found in the bottom of the dump D.

The control information CI is set to Ω. Later, when the interpretation of the p-text-part-list of the body starts, CI is initialized as described in section 8.4.2.

The state component return information RI is initialized with data necessary for the interpretation of a return statement and which are available only at the installation of the body. RI consits of two components:

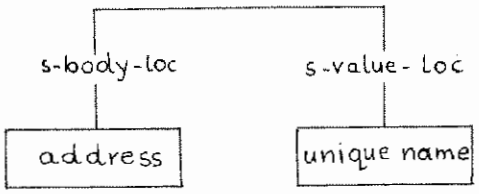



Fig. 21 The return information

The body-loc component holds the address under which the body and its execution status is stored in the procedure body directory P.

The value-loc component specifies the unique name which has been generated when the function reference was encountered and which is used to store the function value into the denotation directory before the body is left.

The following actions are performed now:

Change of the execution status:

To indicate that no other interpretation of the procedure body is allowed till the current interpretation is finished, the execution component of the corresponding entry in the procedure body directory P is set to T.

Installation of the arguments:

For arguments where an address was passed, this address now is connected with the corresponding parameter in an entry into the environment E.

If a value was passed, this value constitutes the value component of the denotation of the corresponding parameter which is constructed now and entered into the denotation directory. The necessary address is the result of the address function af which maps an identifier (in this case the parameter) and scope information (the address of the body within the procedure body directory is used as scope information) onto an address. This address is connected now with the parameter in an entry into the environment.

Interpretation of the p-declaration-part:

The environment is updated by all identifiers declared in the p-declaration-part which are not used as parameters. If a redeclaration of an identifier occurs, the existing old entry in E is overriden.

After that, all necessary entries for the new declared identifiers into the denotation directory are made. In the case of a label only the index-list component of the denotation is initialized with the corresponding declaration. In the case of a builtin declaration only the at component of the denctation is initialized, namely with the elementary object BUILTIN. In the case of a variable the at component of the denotation gets the corresponding declaration, namely INTG or CHAR.

Note that by the definition of the address function af, each interpretation of the procedure will lead, for the locally declared variables, to the same address in DN. The variables have a "static" storage address and their values remain unchanged between executions of the procedure.

Interpretation of the p-text-part-list:

The iuterprettion of a p-text-part-list differs only slightly from the interpretation of a text-part-list. While, on the one hand, a text-part consists of a statement and a text which has to be scanned for possible replacements when the interpretation of the statement is finished, a p-text-part, on the other hand, consists only of a statement which itself must not contain any text. Hence the interpretation of a p-text-part consists only of the interpretation of a statement.

### 8.5.2.3 The return statement

The only way to leave a procedure body and to return control and function value to the place from which the procedure was invoked, is via a return statement. Hence, a proper procedure body must contain at least one return statement within its p-text-part-list. The return statement specifies an expression, which if evaluated and converted, represents the value of the function procedure.

The interpretation of a return statement starts with the evaluation of its expression. Using the return type specified by the body the necessary conversion is done. The result is assigned to the dummy that was generated when the function reference was encountered and which is stored in the value-loc component of RI, i.e., an entry for the dummy in DN is made. The address found in the body-loc component of RI allows the indication in the procedure body directory P, that the interpretation of the procedure is completed and a new reference to it can be made, by altering the corresponding execution component from T to F.

The old state components saved at the beginning of the procedure interpretation in D have to be reinstalled. The function value which is stored in the denotation of the dummy is also available outside the procedure, because the unique name was saved here too (in the control). This value is restored, and the interpretation of the expression or the text that was interrupted by the function reference continues with the value in place of the reference.

## 8.5.3 REFERENCE TO A BUILTIN FUNCTION OCCURRING IN AN EXPRESSION

A reference refers to a builtin function if

(1) the identifier of the reference is the abstract representation of one of the concrete identifiers: INDEX, LENGTH, SUBSTR, and

(2) either the identifier of the reference is not "known", i.e., the identifier has no entry in the current environment E, or the identifier is associated with the denotation of a builtin function name (cf. section 8.2.2).

A reference to a builtin function is proper if the argument list of the reference is proper in its length. In the case the identifier is not known, prior

to the evaluation of the reference a default declaration is performed, giving the identifier the denotation of a builtin function within the current scope.

## 8.6 THE SCAN AND REPLACEMENT MECHANISM

The interpretation of a text-part starts with the interpretation of the contained statement, and continues (if the normal flow of control is supposed) with the interpretation of the corresponding text. The abstract text which is due to be scanned by the replacement mechanism is a list of character values. The grouping in tokens and argument lists, and the necessary check on comments and strings will be done dynamically by the interpreter.

The interpretation of a text starts by copying it out of the corresponding text-part, whereby the character value BLANK added as first element to the copy ensures that in the generated output of the interpreter a blank occurs where in the corresponding input a statement had occurred.

Roughly speaking, the interpretation of the copied text will be done from left to right. Beginning with the leftmost character value and continuing from left to right the interpreter combines a number of consecutive character values (at least one) to a token. Now it is checked whether this token represents a PL/I-identifier. If it does not, the token is transferred into the output medium of the machine and the next token is produced.

This output medium is represented by the state component result cell R. Transfer into R means, that the char-val-list to be transferred is concatenated with the already stored char-val-list in R. When the interpretation of the program is finished the list of character values specified by R is considered to be a concrete PL/I program which can be the input to the translation-interpretation process of PL/I (/4/, /5/).

When a token is encountered that represents a PL/I-identifier, it becomes an expectant for possible replacement, provided that none of the following points turn out to be true:

(1) The corresponding abstract identifier is not "known" by the interpreter, i.e., it has no entry in the current environment.

(2) The corresponding abstract identifier denotes a label.

(3) The rescan component of the denotation of the corresponding abstract identifier is Ω (not activated).

In these cases the token is also transferred into R as described above. What remains are the proper cases for the substitution process:

(1) The identifier denotes an activated variable.

(2) The identifier denotes an activated entry name.

(3) The identifier denotes an activated builtin function name.

In these cases the substitution process for the token under consideration must not necessarily be successful. The important point is, that if any of the various additional conditions is not met, e.g., in the case of a variable the value component of the denotation is Ω, and therefore the substitution cannot be carried out, the token under consideration is not transferred into R, but rather an error is reached.

When the reference to a variable, a function procedure, or a builtin function was successful, i.e., a value was returned, this value is at first converted to a char-val-list, if necessary, and to both ends of the list the character value BLANK is appended, in order to secure the blanks in which the substituted value must be embedded. A case distinction is performed now:

(a) If the replacement value was an integer value before the conversion, or if the rescan component of the denotation of the reference is F, i.e.,

indicating NORESCAN, the char-val-list is transferred to R (instead of the corresponding reference).

(b) If the rescan component of the denotation of the reference is T, the char-val-list is not yet transferred to R, but rather is immediately rescanned for further replacements, as if it were the copy of a text, copied out of a text-part.

Reference to a function procedure:

When the token under consideration turned out to represent an activated entry name of a specified procedure (the body-loc component of the corresponding denotation exists), and the body is proper, a check has to be made whether the procedure possesses parameters. If it is the case, the list of character values following that entry name in the text has to be scanned to get the argument list of the function reference.

An argument is defined to be a char-val-list delimited by the character values COMMA or RIGHT-PAR occurring outside a string. But the char-val-list found inside of matching left-right parenthesis during the scan is not to be searched for these delimiters. The result of this parsing process is a list, each element representing a char-val-list to be used as argument of the function reference.

Provided that the length of that list coincides with that of the parameter list of the body, these argument texts are rescanned for possible replacement. Each char-val-list representing an argument of the function reference is interpreted exactly as if it were a text, with the only difference, that any token that is encountered during this interpretation and which is not replaceable or should not be replaced, is not transferred to R, but rather is transferred into a dummy entry of the denotation directory, which was created when the scan and replacement mechanism started for the argument under consideration. After the rescan is completed for the argument, the char-val-list contained in the dummy, which represents the modified argument, is restored and its necessary conversion to the type required for the parameter will be done.

The further interpretation of the function reference is in analogy with section 8.5.2.2.

For the builtin functions the same mechanism as above is applied to their arguments.

## 8.7 SUMMARY OF THE STATE COMPONENTS AND THEIR PROPERTIES

### 8.7.1 THE EXTERNAL PROGRAM DIRECTORY EP

The external program directory EP contains all external programs which could be incorporated by means of include statements. The main program as well as the external programs satisfy the predicate is-program. (These external programs could be regarded as the output of the translator when applied to the corresponding concrete strings of external text stored in an external library.)

An include statement specifies at least one identifier-pair, where one of the two elements may also be Ω. The corresponding external program is contained in the external text storage under the selector sel(id-pair).

Note that EP is the unique state component that remains unchanged during the entire interpretation process.

### 8.7.2 THE RESULT CELL R

During the interpretation of a program the text is scanned and possible replacements are made. The result is copied in the output medium of the machine. This output medium is represented by the state component R. When the interpretation of the compile time program is terminated, the list of character

values specified by R is considered to be a concrete PL/I program which can be the input to the parsing-translation-interpretation process of PL/I (/4/, /5/).

### 8.7.3 THE UNIQUE NAME COUNTER UN

In two situations during the interpreation of a program unique names are needed, namely for saving the function value in the denotation directory when the interpretation of a procedure is finished and the dump is popped up, and for storing intermediate results into the denotation directory during the scan and replacement process of argument text.

The unique name counter UN has the only purpose to count the unique names already used, thereby guaranteeing that no unique name is used more than once. Whenever a new unique name is needed, the instruction un-name returns one which is different from all unique names used before.

### 8.7.4 THE DENOTATION DIRECTORY DN

The denotation directory associates addresses (in the case of dummies also unique names) with denotations. The denotation of an identifier represents the entire information about it, except of its scope. The different types of denotations are discussed in section 8.2.2.

### 8.7.5 THE PROCEDURE BODY DIRECTORY P

The component P contains the description of procedure bodies. Each entry, which is identified by an address, consists of two components. Access to a procedure body gives an indirect step via the body-loc component of the denotation of the specified entry name.

s-body selects the procedure body;

s-execution specifies the execution status of the corresponding body. The elementary object T specifies that the specific body is just under interpretation. With the aid of this component any attempt to reference a function recursively will be detected and marked as an error.

### 8.7.6 THE ENVIRONMENT E

The environment associates identifiers occurring in a declaration-part or in a p-declaration-part with addresses under which the corresponding denotation is fonnd in the denotation directory. The addresses are generated by the one-to-one function af, mapping scope information ("*" for global scope, or the address of a procedure body) and an identifier onto an address. This function ensures the "static" storage for variables.

Because of the scope rules for the use of identifiers the environment will be updated and changed by two different mechanisms during the interpretation of declarations, depending on whether a declaration-part or a p-declaration-part is to be interpreted.

An interpretation of a declaration-part will appear at the very beginning of the interpretation of a main program, but also during the interpretation of an external program, incorporated by means of an include statement. This kind of updating E does not override existing entries in the environment. Each attempt to do so will be detected and a multiple declarations error will result, because it is not possible to redeclare previously declared identifiers within the declaration-part of an external program.

The interpretation of a p-declaration-part on the other hand is only possible during the interpretation of a function call. Here it is possible to redeclare identifiers to be used in the procedure body with a new meaning. The old

environment must be saved in the dump, to be reinstalled after the completion of the function reference.

### 8.7.7 THE CONTROL INFORMATION CI

The control information CI governs the flow of control through the nested structure of either text-parts and statements outside procedures, or p-text-parts and p-statements inside procedures.

It contains a txt component which usually is a (p-)text-part-list or a (p-)if-statement and an index component which usually is either an integer value or a truth value (T or F). The index identifies a component of the txt component, which is currently beeing interpreted: If the txt component is a (p-)text-part-list, the index is an integer value i identifying the i-th element of the list; if the txt component is a (p-)if-statement, the index is T or F identifying the then or else component, respectively, of the (p-)if-statement.

Whenever during the interpretation of a (p-)statement a (p-)text-part-list or a component of a (p-)if-statement is to be interpreted, by initializing the txt and index components with the corresponding data, the old txt and index components of CI have to be saved. For this purpose the complete control information is stacked into a third component s-ci(CI) of the new control information whenever a new level of nesting is entered.

Similarly, the control C is stacked in a fourth component of the control information, whenever a new level of statement nesting is entered. This control contains the actions to be performed after return form the nested statement level.

The program-name component of the control information exists only if the txt component represents the outermost text-part-list of the main program or of an external program. This component presents the name of the program, i.e., MAIN in the case of the main program, or a pair of identifiers specified by the corresponding include statement in the case of an external program. The program-name component is necessary only for the interpretation of goto statements.

### 8.7.8 THE CONTROL C

The control component of the state is an abstract object which contains a set of instructions to be executed by the machine. The instructions may be considered as arranged in the form of a tree where instructions may have a set of successor instructions and the instructions at the terminal nodes of the tree, i.e., those which have no successor instructions at all, are candiates for immediate execution. For more explicit information about the control and its cooperation with the computation refer to /5/ and /9/.

### 8.7.9 THE DUMP D

At the time of an interpretation of a function reference some information must be saved, to be reinstalled after the completion of the interpretation. The dump, designed as a push down mechanism, holds this information.

There are five components of D:

| | |
|---|---|
| s-e | holds the environment that was active when the function reference was encountered. |
| s-ci | holds the control information which was active when the function reference was encountered. |
| s-c | saves the old control, because a new control must be established for the time of the interpretation of the procedure body. |

s-ri — if the current function reference itself appeared during the interpretation of a procedure body the selector gives access to the return information of the outer procedure.

s-d — holds previous levels of the dump.

### 8.7.10 THE RETURN INFORMATION RI

For the interpretation of a return statement, transferring the control of interpretation back to the place of the function reference, some information is necessary which is available only at the beginning of the interpretation of the procedure. The state component RI will be used to save such information. It consists of two components:

s-value-loc — specifies the unique name that was generated when the function reference was encountered, and to which the function value is assigned before the procedure body is left.

s-body-loc — is used to identify the entry in P, so that its execution component can be altered to F to indicate the completion of the function call and to allow in consequence a new reference to that procedure body.

# 9. THE INTERPRETER

This chapter provides the formal definition of the interpretation of abstract compile time programs. The method used is based on the definition of an abstract machine which is characterized by the set of its states and its state transitions. An abstract compile time program, possibly together with a set of external programs, specifies an initial state of the machine, and the subsequent behaviour of the machine is said to define the interpretation of the compile time program.

The syntax and semantics of the meta language is defined in chapter 1 of /5/.

## 9.1 ABSTRACT SYNTAX OF THE MACHINE STATES

All machine states are objects satisfying the predicate is-state defined below.

(1) is-state =

```
(<s-ep:is-ep>,
 <s-r:is-char-val-list>,
 <s-un:is-intg-val>,
 <s-dn:is-dn>,
 <s-p:is-p>,
 <s-e:is-e>,
 <s-ci:is-ci>,
 <s-c:is-c>,
 <s-d:is-d>,
 <s-ri:is-ri>)
```

### Abbreviations and terms

For the convenience of reference to parts of the state the following abbreviations and terms for components of a given state ξ are introduced and used throughout the interpretation.

| | | |
|---|---|---|
| EP | = s-ep(ξ) | the external program directory |
| R | = s-r(ξ) | the result cell |
| UN | = s-un(ξ) | the unique name counter |
| DN | = s-dn(ξ) | the denotation directory |
| P | = s-p(ξ) | the procedure body directory |
| E | = s-e(ξ) | the environment |
| CI | = s-ci(ξ) | the control information |
| C | = s-c(ξ) | the control |
| D | = s-d(ξ) | the dump |
| RI | = s-ri(ξ) | the return information |

In basic instruction definitions the selectors selecting immediate components of the state are underlined for better readability.

(2) is-ep =

({<sel(idp):is-program> | | is-id-pair(idp)})

Note: The external programs are translated in the same way as the main program, i.e., if txt is the char-val-list denoting a concrete external program,

translate•parse(txt)

is the corresponding abstract program to be stored in the external program directory.

(3) is-dn =

({<ad:is-den> | | is-ad(ad) ∨ is-n(ad)})

Note: Unique names, i.e., is-n(ad), are used only to store intermediate results (is-dummy-den) in the denotation directory.

(4) is-den =

is-var-den ∨ is-entry-den ∨ is-builtin-den ∨ is-label-den ∨ is-dummy-den

(5) is-var-den =

(<s-at:is-prop-var>,
<s-rescan:is-T ∨ is-F ∨ is-Ω>,
<s-value:is-prop-intg-val ∨ is-char-val-list ∨ is-Ω>)

Ref.: is-prop-intg-val 9-19(65)

(6) is-entry-den =

(<s-at:is-ENTRY ∨ is-Ω>,
<s-rescan:is-T ∨ is-F ∨ is-Ω>,
<s-body-loc:is-ad ∨ is-Ω>)

(7) is-builtin-den =

(<s-at:is-BUILTIN>,
<s-rescan:is-T ∨ is-F ∨ is-Ω>)

(8) is-label-den =

(<s-index-list:is-index-list>,
<s-progr-name:is-MAIN ∨ is-id-pair ∨ is-Ω>)

(9) is-dummy-den =

is-prop-intg-val ∨ is-char-val-list

Ref.: is-prop-intg-val 9-19(65)

(10) is-p =

({<ad:is-p-entry> | | is-ad(ad)})

(11) is-p-entry =

(<s-body:is-body>,
<s-execution:is-T ∨ is-F>)

(12) is-e =

({<id:is-ad> | | is-id(id)})

(13) is-ci =

is-Ω ∨
(<s-txt:is-st ∨ is-p-st>,
<s-index:is-index ∨ is-Ω>,
<s-progr-name:is-MAIN ∨ is-id-pair ∨ is-Ω>,
<s-ci:is-ci>,
<s-c:is-c>)

Note: The txt component is a text-part-list or an if-statement (a p-text-part-list or a p-if-statement within procedures). Only in erroneous cases during a goto statement the txt component may be any statement (p-statement within procedures).

The progr-name component exists only if the txt component is the outermost text-part-list of the main program (is-MAIN) or the outermost text-part-list of an external program (is-id-pair).

(14) is-d =

is-Ω ∨
(<s-e:is-e>,
<s-ci:is-ci>,
<s-c:is-c>,
<s-ri:is-ri>,
<s-d:is-d>)

(15) is-ri =

is-Ω ∨
(<s-body-loc:is-ad>,
<s-value-loc:is-n>)

## 9.2 INITIAL STATE AND COMPUTATION OF THE COMPILE TIME MACHINE

The compile time machine describes the interpretation of a compile time program t by defining the set of possible computations resulting from the program. A computation is a sequence of states

$$\xi(0), \xi(1), \xi(2), \ldots$$

satisfying the following two conditions:

(1) $\xi(0)$ is an initial state of t, as given by the function initial-state.

(2) Any two adjacent states $\xi(i), \xi(i+1)$ in the computation must represent a valid step. The validity of a step is defined by the predicate is-step, i.e., any two steps $\xi(i), \xi(i+1)$ in the computation must satisfy the condition

is-step($\xi(i), \xi(i+1)$).

A computation is "successful" if it is finite:

$\xi(0), \xi(1), \ldots, \xi(n)$

and if its end state $\xi(n)$ satisfies the condition

is-Ω•s-c($\xi(n)$).

(16) is-step($\xi$-1,$\xi$-2) =

$(\exists\sigma)(\sigma \in$ term-node•s-c($\xi$-1) & $\xi$-2 = compute($\xi$-1,$\sigma$•s-c))

for: is-state($\xi$)

Ref.: is-state 9-1(1)

Note: Cf. section 1.3.3 of /5/.

(17) initial-state(t,ep) =

is-program(t) & is-ep(ep) →

$\mu_D$(<s-ep:ep>,<s-r:<>>,<s-un:0>,<s-c:int-program(t,MAIN)>)

T → error

Ref.: is-ep 9-2(2)
int-program 9-5(18)

## 9.3 PROGRAM INITIALIZATION

A program initialization is performed at the very beginning of the computation, but also on each incorporation of an external program by means of an include statement (cf. section 9.5.4).

A program initialization comprehends the following actions:

(1) The control information CI and control C are stacked into CI (This is of importance only on initializing an external program).

(2) The text-part-list of the program to be initialized is entered as the txt component of CI, the index component is set to 0.

(3) The program name, i.e., in the case of the main program the object MAIN, in the case of an external program the identifier-pair specified by the corresponding include statement, is entered as the progr-name component of CI. By this, on interpreting a goto statement, the outermost text-part-list of that program can be found in which the label was declared.

(4) The multiple declaration check is performed, i.e., it is tested whether the identifiers declared in the program to be initialized have not been declared previously (trivial in the case of the main program).

(5) The state components E, DN, and P are updated according to the various declarations of the declaration-part.

Metavariables:

| | |
|---|---|
| is-decl-part(dp) | the declaration-part of the program to be initialized |
| (is-MAIN ∨ is-id-pair)(pn) | the program name |
| is-id(id) | |
| is-ad(ad) | |

(18) int-program(t,pn) =

$$\underline{\text{s-ci}}:\mu_0(\langle \text{s-txt}:\text{s-text-part-list}(t)\rangle,\langle \text{s-index}:0\rangle,\langle \text{s-progr-name}:pn\rangle,\langle \text{s-ci}:\underline{CI}\rangle,\langle \text{s-c}:\underline{C}\rangle)$$
$$\underline{\text{s-c}}:\underline{\text{int-next-text-part}}:\ \underline{\text{int-decl-part}}(\text{s-decl-part}(t))$$

for: is-program(t)

Ref.: int-next-text-part 9-7(26)

(19) int-decl-part(dp) =

$$(\forall id)(\neg \text{is-}\Omega\bullet\text{id}(dp) \supset \text{is-}\Omega\bullet\text{id}(\underline{E}) \vee \text{is-entry}\bullet\text{id}(dp) \,\&\, \text{is-entry-den}(den_0) \,\&\, (\text{is-}\Omega\bullet\text{s-entry-decl}\bullet\text{id}(dp) \,\&\, \text{is-}\Omega\bullet\text{s-body-loc}(den_0) \vee \text{is-}\Omega\bullet\text{s-body}\bullet\text{id}(dp) \,\&\, \text{is-}\Omega\bullet\text{s-at}(den_0)))\ \rightarrow$$

  int-declarations(dp);
    upd-e(dp)

$$T \rightarrow \underline{\text{error}}$$

where:
$den_0$ = id(E)(DN)

Ref.: is-entry-den 9-2(6)

(20) upd-e(dp) =

null;
  {upd-id(id,af(*,id)) | ¬is-Ω•id(dp) & is-Ω•id(E)}

Note: The first argument of the address function af represents the scope of the identifier under consideration. The "*" indicates "global".

```
(21)  upd-id(id,ad) =

        s-e:μ(E;<id:ad>)

(22)  int-declarations(dp) =

        null;
          (int-decl(id(E),id(dp)) | ¬is-Ω•id(dp))

(23)  int-decl(ad,decl) =

        is-prop-var(decl) --> s-dn:μ(DN;<s-at•ad:decl>)

        is-index-list(decl) -->

          s-dn:μ(DN;<s-index-list•ad:decl>,<s-progr-name•ad:s-progr-name(CI)>)

        is-body•s-body(decl) -->

          upd-dn(ad,decl);
            upd-p(ad,s-body(decl))

        T --> upd-dn(ad,decl)

      for:is-decl(decl)

(24)  upd-p(ad,body) =

        s-p:μ(P;<s-body•ad:body>,<s-execution•ad:F>)

      for:is-body(body)

(25)  upd-dn(ad,decl) =

        is-body•s-body(decl) -->

          s-dn:μ(DN;<s-at•ad:s-entry-decl(decl)>,<s-body-loc•ad:ad>)

        is-id•s-body(decl) -->

          s-dn:μ(DN;<s-at•ad:s-entry-decl(decl)>,<s-body-loc•ad:af(*,s-body(decl))>)

        is-Ω•s-body(decl) --> s-dn:μ(DN;<s-at•ad:ENTRY>)

      for:is-entry(decl)
```

Note: In the second alternative the declaration does not specify a body, but rather the identifier which is associated in the declaration-part with the proper body. This occurs if a body is associated with more than one entry name.

## 9.4 SEQUENTIAL INTERPRETATION OF TEXT-PARTS

This section defines the sequential flow of control through the nested structure of (p-)text-part-lists. Also some features of the if statement and goto statement are reflected here.

This flow is governed by the control information CI, which contains as its txt component the innermost nested (p-)text-part-list or (p-)if-statement whose

components are currently interpreted. The index component of CI localizes that part of the txt component currently beeing interpreted: If the txt component is a (p-)text-part-list the index is an integer value i pointing to the i-th element of the (p-)text-part-list; if the txt component is a (p-)if-statement the index is a truth value T or F denoting its then or else component.

The nested structure of text-part-lists and if-statements is reflected by the components s-ci(CI) and s-c(CI) which contain the control information and control of the state immediately before entering the current level of the structure; they are to be reinstalled after leaving the current level.

The interpretation of a text-part starts with the interpretation of its statement and continues - on sequential flow of control - with the interpretation of its text. The scan and replacement mechanism for text is defined in section 9.9.

Whenever a (p-)text-part or (p-)if-statement is completed, the instruction int-next-text-part is executed. It increases the index by one, in the case of a (p-)text-part-list which is not yet exhausted, or it returns to the former level by reinstalling the former CI and C.

Metavariables:

| | |
|---|---|
| is-index(indx) | an index localizing a (p-)text-part within a (p-)text-part-list or the then or else component within a (p-)if-statement |
| (is-st ∨ is-p-st)(st) | a (p-)statement to be interpreted |

(26) int-next-text-part =

```
is-intg-val•s-index(CI) & s-index(CI) < length•s-txt(CI) →

  continue;
    upd-index

¬is-Ω(D) & is-Ω•s-ci(CI) → error

T →

  s-ci:s-ci(CI)
  s-c:s-c(CI)
```

Note: The second alternative is reached if the outermost p-text-part-list is exhausted, i.e., no return statement had been executed.

(27) upd-index =

```
s-ci:μ(CI;<s-index:s-index(CI) + 1>)
```

(28) continue =

```
int-next-text-part;
  int-opt-text;
    int-st(take-st(s-index(CI),s-txt(CI)))
```

(29) take-st(indx,st) =

is-list(st) & is-intg-val(indx) & 1 ≤ indx ≤ length(st) → s-st•elem(indx,st)

(is-if-st ∨ is-p-if-st)(st) & is-T(indx) → s-then(st)

(is-if-st ∨ is-p-if-st)(st) & is-F(indx) → s-else(st)

T → error

Note: Applied during the goto statement this function ensures that the index list of the label is correct and that no forbidden gotos into groups are performed.

(30) int-st(st) =

(is-text-part-list ∨ is-p-text-part-list)(st) → int-text-part-list(st)

(is-if-st ∨ is-p-if-st)(st) → int-if-st(st)

is-goto-st(st) → int-goto-st(st)

(is-group ∨ is-p-group)(st) → int-group(st)

is-include-st(st) → int-include-st(st)

is-act-st(st) → int-act-st(st)

is-deact-st(st) → int-deact-st(st)

is-null-st(st) → null

is-assign-st(st) → int-assign-st(st)

is-return-st(st) → int-return-st(st)

Ref.: int-if-st 9-9(34)
int-goto-st 9-10(37)
int-group 9-13(45)
int-include-st 9-15(53)
int-act-st 9-15(55)
int-deact-st 9-16(56)
int-assign-st 9-17(60)
int-return-st 9-32(128)

(31) int-opt-text =

is-Ω(D) ⇒ (is-T ∨ is-F)(s-index(CI)) → null

T → int-text(<BLANK>^s-text•elem(s-index(CI),s-txt(CI)),*)

Ref.: int-text 9-36(142)

Note: Within a procedure body, i.e., ¬is-Ω(D), no text exists; furthermore, the then as well as the else component of an if-statement is a statement rather than a text-part. In these cases this instruction replaces itself by the null instruction. In all other cases the associated text is interpreted by the instruction int-text.

The leading BLANK is inserted to seperate the text from the text of the previously interpreted text-part. The second argument of int-text, namely "*", defines that the resulting output text must be transfered into R in

distinction from the action required during the interpretation of function arguments (cf. int-arg-text in section 9.9).

(32) int-text-part-list(tpl) =

stack-ci(1,tpl)

for:(is-text-part-list ∨ is-p-text-part-list)(tpl)

(33) stack-ci(indx,st) =

s-ci:$\mu_0$(<s-txt:st>,<s-index:indx>,<s-ci:CI>,<s-c:C>)
s-c:continue

## 9.5 INTERPRETATION OF FLOW OF CONTROL STATEMENTS

This section defines the semantics of the if-statement in section 9.5.1, the goto statement in section 9.5.2, the group in section 9.5.3, and the include statement in section 9.5.4.

### 9.5.1 IF STATEMENT

The interpretation of an if-statement comprehends the evaluation of the expression into a truth value and the interpretation of the alternative statement denoted by this truth value by introducing a new level into the control information CI, using the truth value as new index component of CI.

(34) int-if-st(st) =

stack-ci(truth,st);
truth:eval-truth(s-expr(st))

for:(is-if-st ∨ is-p-if-st)(st)

Ref.: stack-ci 9-9(33)

(35) eval-truth(expr) =

pass-truth-val(bs);
bs:pass-convert(BIT,v);
v:eval-expr(expr)

for:is-expr(expr)

Ref.: convert 9-23(93)
eval-expr 9-18(62)

(36) truth-val(bs) =

(∃i)(is-intg-val(i) & 1 ≤ i ≤ lgth(bs) & is-1-BIT•elem(i,bs))

for:is-bit-string(bs)

cont'd

Ref.: lgth 9-21(79)

### 9.5.2 GOTO STATEMENT

In general, the denotation of a label consists of two components: A program name, identifying either the main program or the external program into which the goto shall lead, i.e., the program where the label is declared, and an index list localizing the statement to which the goto shall lead relative to the text-part-list of the program or to the p-text-part-list of a body. Labels which are local to a body do not possess a program name in their denotation.

The interpretation of the goto statement is performed in four steps:

(1) On goto statements outside procedures, the levels of text-part-lists and if-statements reflected in the control information CI are terminated one by one until a level is reached which belongs to the program into which the goto shall lead.

(2) Again, levels of CI are terminated until the statement to which the goto shall lead is contained (possibly at a nested level) in the current txt component of CI.

(3) New levels of text-part-lists and if-statements are established in CI, according to the index list of the label, simulating the situation which would have occurred if these levels would have been entered by the normal flow of control, until the statement to which the goto shall lead is one of the immediate components of the current txt component of CI.

(4) The index component of CI is adjusted so that it denotes the statement to which the goto shall lead.

Metavariables:

is-index-list(indl)

is-ci(ci)

(37) int-goto-st(st) =

$is\text{-}\Omega(ad_0) \vee \neg is\text{-}label\text{-}den(den_0) \vee \neg is\text{-}\Omega(D) \;\&\; \neg is\text{-}\Omega\text{•}s\text{-}progr\text{-}name(den_0) \rightarrow$ error

$\neg is\text{-}\Omega(D) \rightarrow$ goto(s-index-list($den_0$))

$T \rightarrow$ goto-program($den_0$)

where:
$ad_0$ = s-label(st) (E)
$den_0$ = $ad_0$(DN)

for:is-goto-st(st)

Ref.: is-label-den 9-2(8)

Note: A goto leading out of a procedure is erroneous. This is the case if the label denotation contains a program name and the goto statement occurs within a procedure.

(38) goto-program(den) =

is-MAIN•progr-name(CI) & ¬is-MAIN•s-progr-name(den) --• error

progr-name(CI) = s-progr-name(den) --• goto(s-index-list(den))

T --•

s-ci:s-ci(CI)
s-c:goto-program(den)

for:is-label-den(den)

Ref.: is-label-den 9-2(8)

Note: The first alternative is reached if the external program to which the label refers is not incorporated, i.e., is not under interpretation.

(39) progr-name(ci) =

¬is-Ω•s-progr-name(ci) --• s-progr-name(ci)

T --• progr-name•s-ci(ci)

for:¬is-Ω(ci)

Note: This function gives the name of the program to which the top level of the argument belongs.

(40) goto(indl) =

(∃list)(is-index-list(list) & length(list) ≥ 1 & ci-indl(CI)^list = indl) --•

goto-1((ιlist)(is-index-list(list) & ci-indl(CI)^list = indl))

T --•

s-ci:s-ci(CI)
s-c:goto(indl)

(41) ci-indl(ci) =

¬is-Ω•s-progr-name(ci) --• <>

T --• ci-indl-1•s-ci(ci)

(42) ci-indl-1(ci) =

is-Ω(ci) --• <>

¬is-Ω•s-progr-name(ci) --• <s-index(ci)>

T --• ci-indl-1•s-ci(ci)^<s-index(ci)>

Note: The first alternative is reached only inside procedures (no program name exists), the second alternative only outside procedures (a program name exists in any case).

(43) goto-1(indl) =

is-<>•tail(indl) --• goto-2($index_1$)

T --•

s-ci:$\mu_0$(<s-txt:$st_1$>,<s-ci:$\mu$(CI;<s-index:$index_1$>)>,<s-c:int-next-text-part;
int-opt-text>)
s-c:goto-1(tail(indl))

where:
$index_1$ = head(indl)
$st_1$ = take-st($index_1$,s-txt(CI))

Ref.: int-next-text-part 9-7(26)
int-opt-text 9-8(31)
take-st 9-8(29)

(44) goto-2(indx) =

s-ci:$\mu$(CI;<s-index:indx>)
s-c:continue

for:is-index(indx)

Ref.: continue 9-7(28)

### 9.5.3 GROUP

The group corresponds in a concrete program to the iterated group. The interpretation of a group comprehends the following actions: First, the initiation expression is evaluated (as well as the by- and to-expressions) and assigned to the controlling variable. Second, the value of the controlling variable is compared with the value of the to-expression. Third, the (p-)text-part-list is interpreted, and fourth, the iteration is continued by adding the value of the by-expression to the value of the controlling variable and starting a new circle beginning with the second point.

If the to-expression is missing, the comparison is assumed always to yield T. If the by-expression, but not the to-expression is missing, the by-expression is assumed to be the integer value 1. If both, the by-expression and the to-expression are missing, the iteration is not continued.

Metavariables:

| | |
|---|---|
| (is-text-part-list ∨ is-p-text-part-list)(tpl) | the iterated (p-)text-part-list |
| is-id(id) | the controlling variable |
| (is-T ∨ is-F)(truth) | a truth value |

```
(45)  int-group(st) =

        ¬is-Ω(ad0) & is-var-den•ad0(DN) -->

          iterate-do(id0,by-to-vs,s-do-list(st));
            convert-assign(id0,v);
              by-to-vs:eval-by-to(s-by•s-iteration(st),s-to•s-iteration(st));
                v:eval-expr(s-init•s-iteration(st))

        T --> error


      where:
        id0 = s-contr-var•s-iteration(st)
        ad0 = id0(E)


      for:(is-group v is-p-group)(st)


      Ref.:    is-var-den 9-2(5)
               convert-assign 9-18(61)
               eval-expr 9-18(62)


(46)  eval-by-to(by,to) =

        is-Ω(by) & ¬is-Ω(to) -->

          pass(by-to-vs);
            s-by(by-to-vs):pass(1),
            s-to(by-to-vs):eval-expr(to)

        T -->

          pass(by-to-vs);
            s-by(by-to-vs):eval-opt-expr(by),
            s-to(by-to-vs):eval-opt-expr(to)


      for:(is-expr v is-Ω)(by) & (is-expr v is-Ω)(to)


      Ref.:    eval-expr 9-18(62)


(47)  eval-opt-expr(expr) =

        is-Ω(expr) --> PASS:Ω

        is-expr(expr) --> eval-expr(expr)


      Ref.:    eval-expr 9-18(62)


(48)  iterate-do(id,by-to-vs,tpl) =

        do-continue(truth,id,by-to-vs,tpl);
          int-do-list(truth,tpl);
            truth:eval-comp(id,by-to-vs)
```

(49) eval-comp(id,by-to-vs) =

is-Ω•s-to(by-to-vs) → PASS:T

T → PASS:truth-val•eval-infix-expr($v_0$,s-to(by-to-vs),$opr_0$)

where:
$v_0$ = s-value(id(E)(DN))
$opr_0$ = comp-opr•eval-infix-expr(s-by(by-to-vs),0,GE)

Ref.: eval-infix-expr 9-19(68)

(50) comp-opr(bvl) =

truth-val(bvl) → LE

T → GE

for:is-bit-val-list-1(bvl)

(51) int-do-list(truth,tpl) =

¬truth → null

T → int-text-part-list(tpl)

Ref.: int-text-part-list 9-9(32)

(52) do-continue(truth,id,by-to-vs,tpl) =

¬truth ∨ is-Ω(by-to-vs) → null

T →

iterate-do(id,by-to-vs,tpl);
convert-assign(id,eval-infix-expr($v_0$,s-by(by-to-vs),ADD))

where:
$v_0$ = s-value(id(E)(DN))

Ref.: convert-assign 9-18(61)
eval-infix-expr 9-19(68)

### 9.5.4 INCLUDE STATEMENT

The include statement specifies a list of pairs of identifiers, where each pair corresponds to an external program in the external program directory EP. According to this list, the corresponding programs are incorporated and interpreted, one after the other, whereby the pairs of identifiers are used as program names, necessary for the interpretation of goto statements outside procedures.

(53) int-include-st(st) =

int-include(s-id-pair-list(st))

for:is-include-st(st)

(54) int-include(idpl) =

is-<>(idpl) --> null

¬is-Ω(ext-progr$_1$) -->

int-include(tail(idpl));
int-program(ext-progr$_1$,idp$_1$)

T --> error

where:
ext-progr$_1$ = sel(idp$_1$)(EP)
idp$_1$ = head(idpl)

for:is-id-pair-list(idpl)

Ref.: int-program 9-5(18)

## 9.6 ACTIVATE AND DEACTIVATE STATEMENTS

Activate and deactivate statements may appear only outside procedures. By them the activation status of entry names, builtin functions, and global variables is controlled, i.e., they determine whether a reference to a variable or function whithin text is to be replaced by the corresponding value and if, whether the value is rescanned for possible replacements before it is replaced.

The interpretation of an activate statement is simply performed by assigning the rescan component of each activation to the rescan component of the denotation of the corresponding identifier, whereby T denotes RESCAN and F denotes NORESCAN. Note that concrete declare statements outside procedures are translated into activate statements with T as their rescan components.

A deactivate statement sets the rescan components of the denotations of the corresponding identifiers to Ω.

If a builtin function name which does not possess a denotation is activated or deactivated, in addition a declaration is simulated, giving the corresponding identifier the denotation of the builtin function.

Metavariable:

(is-T ∨ is-F ∨ is-Ω)(rescan) — the rescan component of the denotation of an entry name, or builtin function name, or global variable, where T denotes RESCAN, F denotes NORESCAN, and Ω denotes "not activated".

(55) int-act-st(st) =

int-act-deact(s-act-list(st))

for:is-act-st(st)

(56) int-deact-st(st) =

int-act-deact(s-id-list(st))

for:is-deact-st(st)

(57) int-act-deact(list) =

is-<>(list) --> null

¬is-Ω($ad_1$) & ¬is-label-den($den_1$) -->

int-act-deact(tail(list));
upd-rescan($ad_1$,$rescan_1$)

is-Ω($ad_1$) & is-builtin($id_1$) -->

int-act-deact(tail(list));
decl-and-rescan($id_1$,$rescan_1$)

T --> error

where:
$id_1$ = (is-act•head(list) --> s-id•head(list),
T --> head(list))
$ad_1$ = $id_1$(E)
$den_1$ = $ad_1$(DN)
$rescan_1$ = s-rescan•head(list)

for:(is-act-list ∨ is-id-list)(list)

Ref.: is-label-den 9-2(8)
is-builtin 9-28(114)

(58) upd-rescan(ad,rescan) =

s-dn:μ(DN;<s-rescan•ad:rescan>)

for:is-ad(ad)

(59) decl-and-rescan(id,rescan) =

s-e:μ(E;<id:$ad_0$>)
s-dn:μ(DN;<s-at•$ad_0$:BUILTIN>,<s-rescan•$ad_0$:rescan>)

where:
$ad_0$ = af(*,id)

for:is-builtin(id)

Ref.: is-builtin 9-28(114)

## 9.7 ASSIGNMENT STATEMENT, EXPRESSION EVALUATION, AND CONVERSIONS

The first subsection defines the assignment statement. The second subsection defines the evaluation of expressions except of the evaluation of references, which is described in section 9.8. The third subsection describes the conversions between character, bit, and integer data.

Metavariables:

| | | |
|---|---|---|
| integer | is-intg-val | an integer value |
| v,v1,v2 | is-prop-value | a char-val-list, a bit-string, or a proper integer value |
| intg,intg1,intg2 | is-prop-intg-val | an integer value which is proper with regard to an implementation |
| s,s1,s2 | is-char-val-list ∨ is-bit-string | a value, but not an integer value |
| cv1 | is-char-val-list | a value of type CHAR |
| cv,cv1 | is-char-val | a character value |
| bs,bs1,bs2 | is-bit-string | a value of type BIT |
| bv11,bv12 | is-bit-val-list-1 | a value of type BIT, but not the BIT-NULL-STR |
| bv,bv1,bv2 | is-bit-val | a bit value |
| v11,v12 | is-char-val-list ∨ is-bit-val-list | a value of type CHAR or BIT, but not the empty element BIT-NULL-STR |

### 9.7.1 ASSIGNMENT STATEMENT

(60) int-assign-st(st) =

¬is-Ω($ad_0$) & is-var-den•$ad_0$(DN) →

convert-assign(s-lp(st),v);
v:eval-expr(s-rp(st))

T → error

where:
$ad_0$ = s-lp(st)(E)

for:is-assign-st(st)

Ref.: is-var-den 9-2(5)

(61) convert-assign(id,v) =

s-dn:μ(DN;<s-value*(id(E)):convert(s-at($den_0$),v)>)

where:
$den_0$ = id(E)(DN)

for:is-id(id)

9.7.2 EXPRESSION EVALUATION

This section defines the evaluation of all types of expressions except of the evaluation of references, which is defined in section 9.8. The necessary conversions are defined in the next subsection.

Metavariables:

| | | |
|---|---|---|
| expr | is-expr | an expression to be evaluated |
| opr | is-infix-opr ∨ is-prefix-opr | an operator |

(62) eval-expr(expr) =

is-infix-expr(expr) →

pass-eval-infix-expr(v1,v2,s-opr(expr));
v1:eval-expr(s-op-1(expr)),
v2:eval-expr(s-op-2(expr))

is-prefix-expr(expr) →

pass-eval-prefix-expr(v,s-opr(expr));
v:eval-expr(s-op(expr))

is-paren-expr(expr) → eval-expr(s-op(expr))

is-ref(expr) → eval-ref(expr)

is-value(expr) → PASS:intg-test(expr)

Ref.: eval-ref 9-28(112)

(63) intg-test(value) =

is-proper-value(value) → value

T → error

for:is-value(value)

(64) is-proper-value(value) =

is-intg-val(value) ⊃ is-prop-intg-val(value)

for:is-value(value)

(65) is-prop-intg-val(integer) =

abs(integer) < 10 ↑ P

Note: This predicate examines an integer value whether it is proper with regard to an implementation.

(66) P =

Note: This letter denotes the implementation defined precision for integer values.

(67) is-intg-val(P) & P > 0

(68) eval-infix-expr(v1,v2,opr) =

infix-op(convert($tg_0$,v1),convert($tg_0$,v2),opr)

where:
$tg_0$ = target(v1,v2,opr)

(69) target(v1,v2,opr) =

is-arith-opr(opr) → INTG

is-comp-opr(opr) & (is-intg-val(v1) ∨ is-intg-val(v2)) → INTG

is-comp-opr(opr) & is-bit-string(v1) & is-bit-string(v2) → BIT

is-comp-opr(opr) → CHAR

is-bit-opr(opr) → BIT

is-CAT(opr) & is-bit-string(v1) & is-bit-string(v2) → BIT

is-CAT(opr) → CHAR

(70) infix-op(v1,v2,opr) =

is-arith-opr(opr) → arith-op(v1,v2,opr)

is-comp-opr(opr) → comp-op(v1,v2,opr)

is-bit-opr(opr) → bit-op(v1,v2,opr)

is-CAT(opr) → conc(v1,v2)

(71) arith-op(intg1,intg2,opr) =

is-prop-intg-val•arith-op-1(intg1,intg2,opr) → arith-op-1(intg1,intg2,opr)

T → truncate•arith-op-1(intg1,intg2,opr)

for:is-arith-opr(opr)

(72) truncate(integer) =

for:¬is-prop-intg-val(integer)

Note: This implementation defined function maps integer values which are improper with regard to the implementation into proper integer values.

(73) ¬is-prop-intg-val(integer) ⊃ is-prop-intg-val•truncate(integer)

(74) arith-op-1(intg1,intg2,opr) =

is-ADD(opr) → intg1 + intg2

is-SUBTR(opr) → intg1 - intg2

is-MULT(opr) → intg1 . intg2

is-DIV(opr) → trunc(intg1 / intg2)

(75) comp-op(v1,v2,opr) =

is-intg-val(v1) → intg-comp(v1,v2,opr)

T → string-comp(v1,v2,opr)

(76) intg-comp(intg1,intg2,opr) =

opr ∈ {EQ,GE,LE} & intg1 = intg2 ∨ is-NE(opr) & intg1 ≠ intg2 ∨ opr ∈ {GT,GE} & intg1 > intg2 ∨ opr ∈ {LT,LE} & intg1 < intg2 →

<1-BIT>

T → <0-BIT>

for:is-comp-opr(opr)

(77) string-comp(s1,s2,opr) =

truth-to-bit•is-true-comp(s1,s2,opr)

for:is-comp-opr(opr)

Note: The truth value T or F resulting from the comparison operation, performed by is-true-comp, is transformed into a bit string of length one.

(78) is-true-comp(s1,s2,opr) =

is-NE(opr) --> ¬is-true-comp(s1,s2,EQ)

is-GE(opr) --> ¬is-true-comp(s2,s1,GT)

is-LE(opr) --> ¬is-true-comp(s1,s2,GT)

is-LT(opr) --> is-true-comp(s2,s1,GT)

T -->

is-true-comp-1(adjust-string(max-lgth$_0$,s1),adjust-string(max-lgth$_0$,s2),opr)

where:
max-lgth$_0$ = max(lgth(v1),lgth(v2))

for:is-comp-opr(opr)

Note: The interpretation of the operator NE is reduced to the interpretation of EQ, and the interpretation of GE, LE, LT to the interpretation of GT. Both char-val-lists or bit strings are adjusted by the function adjust-string to the same length.

(79) lgth(s) =

is-list(s) --> length(s)

T --> 0

(80) adjust-string(n,s) =

$\underset{i=1}{\overset{n}{\text{LIST}}}$ (i ≤ length(s) --> elem(i,s),

T --> fill-char$_0$)

where:
fill-char$_0$ = (is-char-val-list(s) --> BLANK,
is-bit-string(s) --> 0-BIT)

for:is-intg-val(n) & n > 0

(81) is-true-comp-1(vl1,vl2,opr) =

is-EQ(opr) --> vl1 = vl2

vl1 = vl2 --> F

head(vl1) = head(vl2) --> is-true-comp-1(tail(vl1),tail(vl2),GT)

is-char-val-list(vl1) --> collat•head(vl1) > collat•head(vl2)

is-bit-val-list(vl1) --> bit-num•head(vl1) > bit-num•head(vl2)

for:length(vl1) = length(vl2) & (is-EQ ∨ is-GT)(opr)

(82) collat(cv) =

Note: This implementation defined function maps character values into integer values, denoting the position of a character value in the collating sequence.

(83) is-intg-val•collat(cv) & (cv ≠ cv1 ⊃ collat(cv) ≠ collat(cv1))

(84) truth-to-bit(x) =

is-T(x) → <1-BIT>

is-F(x) → <0-BIT>

(85) bit-op(bs1,bs2,opr) =

$max\text{-}lgth_0 = 0$ → BIT-NULL-STR

T → bit-op-1(adjust-string($max\text{-}lgth_0$,bs1),adjust-string($max\text{-}lgth_0$,bs2),opr)

where:
$max\text{-}lgth_0 = max(lgth(bs1),lgth(bs2))$

for:is-bit-opr(opr)

(86) bit-op-1(bvl1,bvl2,opr) =

$\underset{i=1}{\overset{length(bvl1)}{LIST}}$ single-bit-op(elem(i,bvl1),elem(i,bvl2),opr)

for:length(bvl1) = length(bvl2) & is-bit-opr(opr)

(87) single-bit-op(bv1,bv2,opr) =

is-AND(opr) →

(is-1-BIT(bv1) & is-1-BIT(bv2) → 1-BIT,

T → 0-BIT)

is-OR(opr) →

(is-1-BIT(bv1) ∨ is-1-BIT(bv2) → 1-BIT,

T → 0-BIT)

(88) conc(s1,s2) =

is-BIT-NULL-STR(s1) → s2

is-BIT-NULL-STR(s2) → s1

T → s1^s2

(89) eval-prefix-expr(v,opr) =

prefix-op(convert($tg_0$,v),opr)

where:
$tg_0$ = ((is-PLUS ∨ is-MINUS)(opr) --> INTG,
is-NOT(opr) --> BIT)

for:is-prefix-opr(opr)

(90) prefix-op(v,opr) =

is-PLUS(opr) --> v

is-MINUS(opr) --> -v

is-NOT(opr) --> not-op(v)

for:(is-prop-intg-val ∨ is-bit-string)(v)

(91) not-op(bs) =

is-BIT-NULL-STR(bs) --> bs

T --> . $\overset{\text{length(bs)}}{\underset{i=1}{\text{LIST}}}$ single-not-op•elem(i,bs)

(92) single-not-op(bv) =

is-0-BIT(bv) --> 1-BIT

is-1-BIT(bv) --> 0-BIT

## 9.7.3 CONVERSIONS

This section defines the conversions between the three types of values: INTG, CHAR, BIT. Because of the different semantics of the null strings of character and bit data, the null string of bit data is the elementary object BIT-NULL-STR, rather than the empty list <>.

(93) convert(da,v) =

type(v) = da --> v

is-intg-val(v) & is-CHAR(da) --> intg-char-conv(v)

is-intg-val(v) & is-BIT(da) --> intg-bit-conv•abs(v)

is-char-val-list(v) & is-INTG(da) --> char-intg-conv(v)

is-char-val-list(v) & is-BIT(da) --> char-bit-conv(v)

is-bit-string(v) & is-INTG(da) --> bit-intg-conv(v)

is-bit-string(v) & is-CHAR(da) --> bit-char-conv(v)

for:(is-INTG ∨ is-CHAR ∨ is-BIT)(da)

(94) type(v) =

is-intg-val(v) --• INTG

is-char-val-list(v) --• CHAR

is-bit-string(v) --• BIT

(95) intg-char-conv(intg) =

intg ≥ 0 --• blank-fill•intg-char(intg)

T --• blank-fill(<MINUS>^intg-char(-intg))

(96) intg-char(intg) =

intg < 10 --• <num-char(intg)>

T --• intg-char•trunc(intg / 10)^<num-char•modulo(intg,10)>

for:intg ≥ 0

(97) num-char(intg) =

intg = 0 --• 0-CHAR

intg = 1 --• 1-CHAR

intg = 2 --• 2-CHAR

intg = 3 --• 3-CHAR

intg = 4 --• 4-CHAR

intg = 5 --• 5-CHAR

intg = 6 --• 6-CHAR

intg = 7 --• 7-CHAR

intg = 8 --• 8-CHAR

intg = 9 --• 9-CHAR

(98) blank-fill(cvl) =

$\underset{i=1}{\overset{P+3}{\text{LIST}}}$ (i ≤ $\text{blank-no}_0$ --• BLANK,

T --• elem(i - $\text{blank-no}_0$,cvl))

where:
$\text{blank-no}_0$ = P + 3 - length(cvl)

for:length(cvl) ≤ P + 3

Note: A char-val-list which is the result of an integer-character conversion must be of length P + 3.

```
(99)  intg-bit-conv(intg) =
        intg < 2 --> <num-bit(intg)>
        T --> intg-bit-conv•trunc(intg / 2)^<num-bit•modulo(intg,2)>
      for:intg ≥ 0

(100) num-bit(intg) =
        intg = 0 --> 0-BIT
        intg = 1 --> 1-BIT

(101) char-intg-conv(cvl) =
        is-prop-intg-val•char-intg(cvl) --> char-intg(cvl)
        T --> truncate•char-intg(cvl)

(102) char-intg(cvl) =
        char-intg-1(cvl,Ω)

(103) char-intg-1(cvl,x) =
        is-BLANK-list(cvl) & ¬is-intg-val(x) --> error
        is-BLANK-list(cvl) --> x
        is-BLANK•head(cvl) & is-Ω(x) --> char-intg-1(tail(cvl),x)
        head(cvl) ∈ {PLUS,MINUS} & is-Ω(x) --> char-intg-1(tail(cvl),head(cvl))
        is-digit•head(cvl) & ¬is-intg-val(x) -->
          char-intg-1(tail(cvl),sign_Ω . char-num•head(cvl))
        is-digit•head(cvl) -->
          char-intg-1(tail(cvl),10 . x + sign(x) . char-num•head(cvl))
        T --> error
      where:
        sign_Ω = (is-MINUS(x) --> -1,
                  T --> 1)

      for:(is-intg-val ∨ is-Ω ∨ is-PLUS ∨ is-MINUS)(x)
```

Note: The char-val-list to be converted must have the form of a possibly signed sequence of digits, optionally surrounded by blanks.

(104) char-num(cv) =

is-0-CHAR(cv) --> 0

is-1-CHAR(cv) --> 1

is-2-CHAR(cv) --> 2

is-3-CHAR(cv) --> 3

is-4-CHAR(cv) --> 4

is-5-CHAR(cv) --> 5

is-6-CHAR(cv) --> 6

is-7-CHAR(cv) --> 7

is-8-CHAR(cv) --> 8

is-9-CHAR(cv) --> 9

(105) char-bit-conv(cvl) =

is-<>(cvl) --> BIT-NULL-STR

T --> $\overset{\text{length(cvl)}}{\underset{i=1}{\text{LIST}}}$ char-bit•elem(i,cvl)

(106) char-bit(cv) =

is-0-CHAR(cv) --> 0-BIT

is-1-CHAR(cv) --> 1-BIT

T --> error

(107) bit-intg-conv(bs) =

is-prop-intg-val•bit-intg(bs) --> bit-intg(bs)

T --> truncate•bit-intg(bs)

(108) bit-intg(bs) =

is-BIT-NULL-STR(bs) --> 0

T --> $\overset{\text{length(bs)}}{\underset{i=1}{\sum}}$ bit-num•elem(i,bs) . 2 ↑ (length(bs) - i)

(109) bit-num(bv) =

is-0-BIT(bv) --> 0

is-1-BIT(bv) --> 1

(110) bit-char-conv(bs) =

is-BIT-NULL-STR(bs) --> <>

T --> $\underset{i=1}{\overset{length(bs)}{LIST}}$ bit-char•elem(i,bs)

(111) bit-char(bv) =

is-0-BIT(bv) --> 0-CHAR

is-1-BIT(bv) --> 1-CHAR

## 9.8 EVALUATION OF REFERENCES

This section defines the evaluation of the three different types of references which may appear in an expression:

Reference to a variable,

reference to a procedure (9.8.1),

reference to a builtin function (9.8.3).

Subsection 9.8.2 defines the interpretation of procedure bodies, independently from the context in which the corresponding reference occurred, i.e., also procedures invoked from text refer to this section. The same holds for the function eval-builtin defined in 9.8.3.

Metavariables:

| | | |
|---|---|---|
| i,j | is-intg-val | integer values |
| id | is-id | an identifier |
| parl | is-id-list | the parameter list of the body |
| dp | is-p-decl-part | the declaration-part of the body |

(112) eval-ref(ref) =

is-<>•s-arg-list(ref) & ¬is-Ω(ad$_0$) & ¬is-Ω•s-value(den$_0$) --> PASS:s-value(den$_0$)

¬is-Ω(ad$_0$) & ¬is-Ω(body-loc$_0$) & ¬is-Ω(body$_0$) & ¬execution$_0$ &
length•s-arg-list(ref) = length•s-param-list(body$_0$) & is-prop-body(body$_0$) -->

restore(n);
call-proc(body-loc$_0$,argl,n);
n:un-name,
argl:eval-arg-list(s-arg-list(ref),s-param-list(body$_0$),
s-decl-part(body$_0$))

is-builtin(id$_0$) & (is-Ω(ad$_0$) ∨ is-builtin-den(den$_0$)) &
is-prop-builtin-argl-length(id$_0$,s-arg-list(ref)) -->

pass-eval-builtin(id$_0$,argl);
argl:eval-builtin-arg-list(s-arg-list(ref),builtin-descrl(id$_0$));
def-decl(id$_0$)

T --> error

cont'd

where:
$id_0$ = s-id(ref)
$ad_0$ = $id_0$(E)
$den_0$ = $ad_0$(DN)
$body\text{-}loc_0$ = s-body-loc($den_0$)
$body_0$ = s-body•$body\text{-}loc_0$(P)
$execution_0$ = s-execution•$body\text{-}loc_0$(P)

for:is-ref(ref)

Ref.: is-builtin-den 9-2(7)

### 9.8.1 REFERENCE TO A PROCEDURE

(113) is-prop-body(body) =

$$(\forall i)(1 \leq i \leq length(parl_0) \supset is\text{-}prop\text{-}var(elem(i,parl_0)(dp_0)) \;\&\; \neg(\exists j)(1 \leq j \leq length(parl_0) \;\&\; i \neq j \;\&\; elem(i,parl_0) = elem(j,parl_0))) \;\&\; (\forall id)(is\text{-}BUILTIN \bullet id(dp_0) \supset is\text{-}builtin(id))$$

where:
$parl_0$ = s-param-list(body)
$dp_0$ = s-decl-part(body)

for:is-body(body)

Note: This predicate performs a syntax check on the body which had not been carried out by the translator.

(114) is-builtin(id) =

id ∈ {mk-id(SUBSTR),mk-id(LENGTH),mk-id(INDEX)}

(115) eval-arg-list(expr-list,parl,dp) =

is-<>(expr-list) → PASS:<>

T →

mk-list(arg,argl);
argl:eval-arg-list(tail(expr-list),tail(parl),dp);
arg:eval-arg(head(expr-list),head(parl)(dp))

for:is-expr-list(expr-list)

(116) eval-arg(expr,da) =

```
is-ref(expr) & is-<>•s-arg-list(expr) & ¬is-Ω(ad0) & s-at(den0) = da →

  PASS:ad0

T →

  pass-convert(da,v);
    v:eval-expr(expr)
```

where:
$ad_0$ = s-id(expr)(E)
$den_0$ = $ad_0$(DN)

for:is-expr(expr) & (is-INTG ∨ is-CHAR)(da)

Ref.: convert 9-23(93)
eval-expr 9-18(62)

(117) un-name =

```
PASS:elem(UN)
s-un:UN + 1
```

(118) restore(n) =

```
PASS:n(DN)
```

for:is-n(n)

## 9.8.2 INTERPRETATION OF THE PROCEDURE BODY

Function references encountered in expressions as well as function references encountered within a text refer to this subsection. In both cases the arguments of the reference are already evaluated, i.e., the argument list consists of addresses possessing the denotation of a variable, and of values of the type INTG or CHAR.

### 9.8.2.1 Initialization

Here, all initializing actions are defined: The execution status of the body is set to T to prevent recursive invocations of the body, a new level of the dump D is introduced, the outermost environment, i.e., the environment where the procedure was declared, is assigned to E, the return information is constructed, the argument list is installed, and at last the declaration-part of the body is interpreted.

Metavariables:

| | | |
|---|---|---|
| body-loc | is-ad | the address under which the body is stored in the procedure body directory P |
| ad | is-ad | the address of an identifier |
| argl | | the argument list; its elements are addresses, proper integer values and char-val-lists |

(119) call-proc(body-loc,argl,n) =

s-p:μ(P;<s-execution•body-loc:T>)
s-e:$e_0$
s-ci:Ω
s-c:int-text-part-list($tpl_0$);
int-p-declarations($dp_0$,$parl_0$);
upd-p-e($dp_0$,$parl_0$);
install-arg-list(argl,$parl_0$)
s-d:stack(E,CI,C,D,RI)
s-ri:$\mu_0$(<s-body-loc:body-loc>,<s-value-loc:n>)

where:
$e_0$ = (is-Ω(D) → E,
T → get-e(D))
$body_0$ = s-body•body-loc(P)
$dp_0$ = s-decl-part($body_0$)
$tpl_0$ = s-text-part-list($body_0$)
$parl_0$ = s-param-list($body_0$)

for:is-n(n)

Ref.: int-text-part-list 9-9(32)

(120) get-e(d) =

is-Ω•s-d(d) → s-e(d)

T → get-e•s-d(d)

for:is-d & ¬is-Ω(d)

Ref.: is-d 9-3(14)

Note: This function yields the outermost environment, i.e., the environment stored at the bottom of the dump.

(121) stack(e,ci,c,d,ri) =

$\mu_0$(<s-e:e>,<s-ci:ci>,<s-c:c>,<s-d:d>,<s-ri:ri>)

(122) install-arg-list(argl,parl) =

null;
(install-arg(elem(i,argl),elem(i,parl)) | 1 ≤ i ≤ length(argl))

(123) install-arg(arg,par) =

is-ad(arg) --→ upd-id(par,arg)

T --→

dummy-assign($ad_1$,arg);
upd-id(par,$ad_1$)

where:
$ad_1$ = af($body\text{-}loc_1$,par)
$body\text{-}loc_1$ = s-body-loc(RI)

for:(is-ad ∨ is-prop-intg-val ∨ is-char-val-list)(arg) & is-id(par)

Ref.: upd-id 9-6(21)
is-prop-intg-val 9-19(65)

Note: The body-location, i.e., the address of the procedure body within the procedure body directory P, serves as scope information for the address function af.

(124) dummy-assign(ad,v) =

s-dn:μ(DN;<s-at•ad:type(v)>,<s-value•ad:v>)

for:(is-prop-intg-val ∨ is-char-val-list)(v)

Ref.: type 9-24(94)
is-prop-intg-val 9-19(65)

(125) upd-p-e(dp,parl) =

null;
{upd-id(id,$ad_1$) | ¬is-Ω•id(dp) &
¬(∃i)(1 ≤ i ≤ length(parl) & id = elem(i,parl))}

where:
$ad_1$ = af($body\text{-}loc_1$,id)
$body\text{-}loc_1$ = s-body-loc(RI)

Ref.: upd-id 9-6(21)

(126) int-p-declarations(dp,parl) =

null;
{int-p-decl(id(E),id(dp)) | ¬is-Ω•id(dp) &
¬(∃i)(1 ≤ i ≤ length(parl) & id = elem(i,parl))}

(127) int-p-decl(ad,decl) =

(is-prop-var ∨ is-BUILTIN)(decl) → s-dn:μ(DN;<s-at•ad:decl>)

is-index-list(decl) → s-dn:μ(DN;<s-index-list•ad:decl>)

for:is-p-decl(decl)

9.8.2.2 Return statement

This section defines the return from the procedure by means of a return statement. The value of the specified expression is converted to the return type of the body and enters the denotation directory DN under the unique name specified by the return information RI, where it is then available outside the procedure activation.

Abbreviations:

| | |
|---|---|
| value-loc$_0$ = s-value-loc(RI) | the unique name under which the function value is entered into DN |
| body-loc$_0$ = s-body-loc(RI) | the address of the body within P |
| ret-type$_0$ = s-ret-type•s-body•body-loc$_0$(P) | the return type of the body |

(128) int-return-st(st) =

upstack;
deact-body;
store(value-loc$_0$,v-1);
v-1:pass-convert(ret-type$_0$,v);
v:eval-expr(s-expr(st))

for:is-return-st(st)

Ref.: convert 9-23(93)
eval-expr 9-18(62)

(129) store(n,v) =

s-dn:μ(DN;<n:v>)

for:is-n(n) & (is-prop-intg-val ∨ is-char-val-list)(v)

Ref.: is-prop-intg-val 9-19(65)

(130) deact-body =

s-p:μ(P;<s-execution•body-loc$_0$:F>)

Note: The execution status associated wich&the&body is set to F, indicating that the interpretation of the body is completed and further references to the body can be made.

(131) unstack =

s-e:s-e(D)
s-ci:s-ci(D)
s-c:s-c(D)
s-d:s-d(D)
s-ri:s-ri(D)

### 9.8.3 REFERENCE TO A BUILTIN FUNCTION

This section defines the evaluation of references appearing in an expression to the builtin functions INDEX, LENGTH, and SUBSTR.

If a proper reference to a builtin function occurs and the corresponding identifier is not known, i.e., has no entry in the environment E, a default declaration is performed, giving the identifier the denotation of a builtin function within the current scope.

The function eval-builtin, whose arguments are the name of the builtin function and the already evaluated and converted list of arguments, yields the value of the corresponding reference. It is also used if the reference occurs in the context of a text.

Metavariables:

| | | |
|---|---|---|
| id | is-builtin | the identifier of a reference to a builtin function |
| expr-list | is-expr-list | the argument list of a reference |
| argl | | the evaluated argument list whose elements are char-val-lists or proper integer values |
| descrl | | the description list whose elements are the elementary objects CHAR and INTG. It is given by the function builtin-descrl. |
| cvl,cvl1,cvl2 | is-char-val-list | |

(132) is-prop-builtin-argl-length(id,expr-list) =

id = mk-id(INDEX) → length(expr-list) = 2

id = mk-id(LENGTH) → length(expr-list) = 1

id = mk-id(SUBSTR) → 2 ≤ length(expr-list) ≤ 3

(133) def-decl(id) =

¬is-Ω•id(E) → null

T →

s-dn:μ(DN;<s-at•$ad_1$:BUILTIN>)
s-e:μ(E;<id:$ad_1$>)

where:
$ad_1$ = af($scope_1$,id)
$scope_1$ = (is-Ω(D) → *,
T → s-body-loc(RI))

Note: Inside a procedure the default declaration holds only during the current procedure activation.

(134) builtin-descrl(id) =

```
id = mk-id(INDEX) --> <CHAR,CHAR>

id = mk-id(LENGTH) --> <CHAR>

id = mk-id(SUBSTR) --> <CHAR,INTG,INTG>
```

(135) eval-builtin-arg-list(expr-list,descrl) =

```
is-<>(expr-list) --> PASS:<>

T -->

  mk-list(arg,argl);
    argl:eval-builtin-arg-list(tail(expr-list),tail(descrl));
      arg:eval-builtin-arg(head(expr-list),head(descrl))
```

(136) eval-builtin-arg(expr,da) =

```
pass-convert(da,v);
  v:eval-expr(expr)

for:is-expr(expr) & (is-INTG v is-CHAR)(da)
```

Ref.: convert 9-23(93)
eval-expr 9-18(62)

(137) eval-builtin(id,argl) =

```
id = mk-id(INDEX) --> eval-index(arg1,arg2)

id = mk-id(LENGTH) --> eval-length(arg1)

id = mk-id(SUBSTR) --> eval-substr(arg1,arg2,arg3)

where:
   arg1 = elem(1,argl)
   arg2 = elem(2,argl)
   arg3 = elem(3,argl)
```

(138) eval-index(cvl1,cvl2) =

```
is-prop-intg-val•index(cvl1,cvl2) --> index(cvl1,cvl2)

T --> truncate•index(cvl1,cvl2)
```

Ref.: is-prop-intg-val 9-19(65)
truncate 9-20(72)

(139) index(cvl1,cvl2) =

is-{}(i-set$_1$) --> 0

T --> min-set(i-set$_1$)

where:
i-set$_1$ = {i | i > 0 & cvl2 = eval-substr(cvl1,i,length(cvl2))}

(140) eval-length(cvl) =

is-prop-intg-val•length(cvl) --> length(cvl)

T --> truncate•length(cvl)

Ref.: is-prop-intg-val 9-19(65)
truncate 9-20(72)

(141) eval-substr(cvl,i,j) =

$$\underset{n=i_1}{\overset{l_1}{\text{LIST}}} \text{elem}(n,\text{cvl})$$

where:
$i_1$ = max(1,i)
$l_1$ = (is-Ω(j) --> length(cvl),
T --> min(length(cvl),$e_1$))
$e_1$ = i + j - 1

for:(is-intg-val v is-Ω)(j)

## 9.9 THE SCAN AND REPLACEMENT MECHANISM

The interpretation of a text-part (cf.section 9.4) starts with the interpretation of its statement and continues - if sequential flow of control is supposed - with the interpretation of the associated text. The interpretation of text is defined in this section. The instruction int-opt-text of section 9.4 constitutes the entry point to this section.

Metavariables:

| | | |
|---|---|---|
| cvl,token,list,arg | is-char-val-list | a text |
| loc | is-* v is-n | denotes the location to which the text is to be transferred after the termination of the replacement process |
| n | is-intg-val | |
| argl | is-char-val-list-list | the argument list of a reference occurring within a text |
| v | is-char-val-list v is-prop-intg-val | the value of a function or variable |

(142) int-text(cvl,loc) =

```
is-<>(cvl) --> null

T -->

  int-text(cvl-1,loc);
    cvl-1:int-token(cvl,loc)
```

Note: The second argument specifies whether a token which is not further replaceable or the value of a function a variable which should not be scanned for replacement is to be transferred into the result cell R (is-*(loc)), or is part of an argument of a (builtin) function reference (is-n(loc)) which must be stored as an intermediate result in the denotation directory DN, using the unique name loc as selector.

(143) int-token(cvl,loc) =

```
is-identifier(token1) --> int-id(token1,tail1,loc)

T -->

  pass(tail1);
    transfer(token1,loc)
```

where:
$token_1$ = s-token•find-token(cvl)
$tail_1$ = s-tail•find-token(cvl)

Note: Only tokens having the form of a PL/I identifier become candidates for the replacement process.

(144) is-identifier(cvl) =

```
is-letter•elem(1,cvl) & is-alpham-char-list(cvl)
```

(145) find-token(cvl) =

```
find-token-1(cvl,<>,Ω)
```

(146) find-token-1(cvl,token,ind) =

```
is-<>(cvl) & is-Ω(ind) --> μ0(<s-token:token>,<s-tail:<>>)

is-<>(cvl) v is-COMMENT(ind) & length(cvl) < 2 --> error

is-Ω(ind) & cvl-begins-with-id0 & is-<>(token) -->

  μ0(<s-token:id0>,<s-tail:tail0>)

is-Ω(ind) & cvl-begins-with-id0 & is-delimiter•last(token) -->

  μ0(<s-token:token>,<s-tail:cvl>)

is-Ω(ind) & length(cvl) > 1 & is-SLASH•head(cvl) & is-ASTER•elem(2,cvl) -->

  find-token-1(tail•tail(cvl),token^<SLASH,ASTER>,COMMENT)

is-COMMENT(ind) & is-ASTER•head(cvl) & is-SLASH•elem(2,cvl) -->

  find-token-1(tail•tail(cvl),token^<ASTER,SLASH>,Ω)

T --> find-token-1(tail(cvl),token^<head(cvl)>,ind1)
```

where:

$cvl\text{-}begins\text{-}with\text{-}id_0 = (\exists n)(length\text{-}of\text{-}id_n)$

$length\text{-}of\text{-}id_n = n \leq length(cvl) \;\&\; is\text{-}identifier(\underset{i=1}{\overset{n}{LIST}}\; elem(i,cvl)) \;\&$

$(is\text{-}\Omega \vee is\text{-}delimiter)(elem(n+1,cvl))$

$id_0 = \underset{i=1}{\overset{n_0}{LIST}}\; elem(i,cvl)$

$n_0 = (\cup n)(length\text{-}of\text{-}id_n)$

$tail_0 = (\cup list)(id_0 \hat{}\, list = cvl)$

$ind_1 = (is\text{-}\Omega(ind) \;\&\; is\text{-}APOSTR•head(cvl) \rightarrow STRING,$
$is\text{-}STRING(ind) \;\&\; is\text{-}APOSTR•head(cvl) \rightarrow \Omega,$
$T \rightarrow ind)$

for: (is-Ω v is-STRING v is-COMMENT)(ind)

Note: This function yields an object with the structure

(<s-token:is-char-val-list>,<s-tail:is-char-val-list>)

where the token component is the first "token" of the text to be scanned; the tail component is the remaining text. A "token" is either an identifier outside strings and comments immediately surrounded by PL/I delimiters within the text, or any substring of the text delimited by those identifiers.

The function also checks the text for unmatched comment or character-string delimiters.

(147) is-delimiter(x) =

x ∈
{PLUS,MINUS,ASTER,SLASH,GT,EQ,LT,NOT,AND,OR,LEFT-PAR,RIGHT-PAR,COMMA,
SEMIC,COLON,BLANK,APOSTR,POINT,PERC}

```
(148)  transfer(token,loc) =

          is-*(loc) --> s-r:R^token

          is-n(loc) --> s-dn:μ(DN;<loc:loc(DN)^token>)


(149)  int-id(identifier,cvl,loc) =

          is-Ω(ad0) v is-Ω(rescan0) -->

            pass(cvl);
              transfer(identifier,loc)

          is-var-den(den0) & ¬is-Ω(value0) -->

            pass(cvl);
              int-value(value0,rescan0,loc)

          is-entry-den(den0) & ¬is-Ω(body-loc0) & ¬is-Ω(body0) & is-prop-body(body0) &
          (is-<>(parl0) v length(parl0) = length(argl1) & is-LEFT-PAR•head(cvl)) -->

            pass(tail2);
              int-value(v,rescan0,loc);
                v:restore(n);
                  call-proc(body-loc0,argl,n);
                    n:un-name,
                    argl:eval-arg-text-list(argl2,parl0,dp0)

          is-builtin(id0) & is-builtin-den(den0) &
          is-prop-builtin-argl-length(id0,argl1) & is-LEFT-PAR•head(cvl) -->

            pass(tail1);
              int-value(v,rescan0,loc);
                v:pass-eval-builtin(id0,argl);
                  argl:eval-builtin-arg-text-list(argl1,builtin-descrl(id0))

          T --> error


       where:
          id0 = mk-id(identifier)
          ad0 = id0(E)
          den0 = ad0(DN)
          rescan0 = s-rescan(den0)
          value0 = s-value(den0)
          body-loc0 = s-body-loc(den0)
          body0 = s-body•body-loc0(P)
          parl0 = s-param-list(body0)
          dp0 = s-decl-part(body0)
          argl2 = (is-<>(parl0) --> <>,
                   T --> argl1)
          argl1 = s-arg-list•arg-parse•tail(cvl)
          tail2 = (is-<>(parl0) --> cvl,
                   T --> tail1)
          tail1 = s-tail•arg-parse•tail(cvl)


       for:is-identifier(identifier)


       Ref.:    is-var-den 9-2(5)
                is-entry-den 9-2(6)
                is-prop-body 9-28(113)
                restore 9-29(118)
                call-proc 9-30(119)
                un-name 9-29(117)
                is-builtin 9-28(114)
```

cont'd

is-builtin-den 9-2(7)
is-prop-builtin-arg1-length 9-33(132)
eval-builtin 9-34(137)
builtin-descr1 9-34(134)

Note: The arguments of this instruction are: The text of an identifier which is candidate for replacement (in the case of a function reference the entire reference is replaced by its value), the remaining text following the identifier which contains the argument list in the case of a function reference, and the location to which the result of the replacement process is to be transferred.

(150) int-value(v,rescan,loc) =

is-intg-val(v) → transfer(app-blanks•convert(CHAR,v),loc)

rescan → int-text(app-blanks(v),loc)

T → transfer(app-blanks(v),loc)

for:(is-T ∨ is-F)(rescan)

Ref.: convert 9-23(93)

Note: This instruction decides whether the value of a reference - if it is of type CHAR - is scanned for possible replacement,dependent on the truth value of the rescan component of the corresponding denotation. In any case, to both ends of the replacement value which is a char-val-list BLANK's are appended.

(151) app-blanks(cvl) =

<BLANK>^cvl^<BLANK>

(152) arg-parse(cvl) =

arg-parse-1(<>,<>,cvl,0,F)

(153) arg-parse-1(arg,argl,cvl,pcount,string) =

```
is-<>(cvl) --> error

is-RIGHT-PAR•head(cvl) & pcount = 0 & ¬string -->

  μ0(<s-arg-list:argl^<arg>>,<s-tail:tail(cvl)>)

is-COMMA•head(cvl) & pcount = 0 & ¬string -->

  arg-parse-1(<>,argl^<arg>,tail(cvl),0,F)

T --> arg-parse-1(arg^<head(cvl)>,argl,tail(cvl),pcount1,string1)
```

where:

$pcount_1$ = (is-LEFT-PAR•head(cvl) & ¬string --> pcount + 1,
is-RIGHT-PAR•head(cvl) & ¬string --> pcount - 1,
T --> pcount)

$string_1$ = (is-APOSTR•head(cvl) --> ¬string,
T --> string)

for:is-intg-val(pcount) & (is-T ∨ is-F)(string)

Note: This function yields an object with the structure

(<s-arg-list:is-char-val-list-list>,<s-tail:is-char-val-list>)

where the arg-list component is the desired argument list of the reference; the tail component is the remaining text.

(154) eval-arg-text-list(argl,parl,dp) =

```
is-<>(argl) --> PASS:<>

T -->

  mk-list(arg,argl-1);
    argl-1:eval-arg-text-list(tail(argl),tail(parl),dp);
      arg:eval-arg-text(head(argl),head(parl)(dp))
```

for:is-id-list(parl) & is-p-decl-part(dp)

(155) eval-arg-text(arg,da) =

```
pass-convert(da,cvl);
  cvl:int-arg-text(arg)
```

for:(is-INTG ∨ is-CHAR)(da)

Ref.: convert 9-23(93)

(156) int-arg-text(arg) =

```
restore(n);
  int-text(arg,n);
    store(n,<>);
      n:un-name
```

cont'd

Ref.: restore 9-29(118)
store 9-32(129)
un-name 9-29(117)

Note: Each char-val-list representing an argument is scanned for possible replacements, using the instruction int-text again. At that point it is necessary to indicate by the second argument of int-text that the result of the replacement process must not be transferred into R, but is rather stored as an intermediate result into DN using the unique name n as selector.

(157) eval-builtin-arg-text-list(argl,descrl) =

```
is-<>(argl) → PASS:<>

T →

  mk-list(arg,argl-1);
    argl-1:eval-builtin-arg-text-list(tail(argl),tail(descrl));
      arg:eval-arg-text(head(argl),head(descrl))
```

## APPENDIX: CROSS-REFERENCE INDEX

This index lists all names used in the document, with the exception of:

Names defined in section 3.2 (concrete syntax),
names defined in chapter 5 (abstract representation of concrete syntax),
names of selectors (i.e., names prefixed by s-),
names of abbreviations and metavariables.

Selectors to state components in basic instructions are referenced however. Formulas are referenced by the form x-yy(zzz) , where x is the number of a main chapter, yy is the page number within the main chapter and zzz is the number of the formula within the main chapter. The following conventions hold:

(1) For all names all instances of use in a formula are given. The defining formula is indicated by an underlined reference.

(2) A function name (but not a selector name) or an instruction name whose defining formula is not specified is defined in chapter 1 of /5/.

(3) Occurrences of names of the form pass-f, where f is a function name, are listed under the entry f.

Occurrences of names of the form is-pred-suffix, where suffix stands for list, list-1 or set, or one of these followed again by - suffix, are listed under the entry is-pred.

Occurrences of names of the form is-OBJ are listed under the entry OBJ.

2 APPENDIX: CROSS-REFERENCE INDEX