TECHNICAL REPORT

TR 25.080
28 June 1968

FORMAL DEFINITION OF THE
PL/I COMPILE TIME FACILITIES

M. FLECK
E. NEUHOLD

IBM LABORATORY VIENNA
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 25,080
28 June 1968
THE INDIAN NATIONAL WEATHER SERVICE

OFFICIAL WEATHER MANUAL

Volume 1

Department of Commerce

United States Government Printing Office

WASHINGTON, D.C. 20402

[Page continues on the next page]
This document is part of a series of documents which represent the formal definition of syntax and semantics of PL/I issued by 28 June 1968:

/1/ LUCAS, P., LAUER, P., STIGLEITNER, H.: Method and Notation for the Formal Definition of Programming Languages.


IBM Laboratory Vienna, Techn. Report TR 25.084.


The method and notation for 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.

An outline of the method is given in /1/, which 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 inside and outside IBM.

The language defined in this present version is PL/I as specified in the official PL/I Language Specifications Form No. Y33-6003 with the exception of the following features which are not included:

- Optimizing attributes (they are included in the concrete syntax but not in the abstract syntax; they are only tested for compatibility with other attributes and used for implication of default attributes),
- Implicit conversion between offsets and pointers,
- The REFSR option,
- The implicit rules for ordering initializing actions in the prologues of blocks and procedures.

The draft for this document was completed by 31 January 1968. It has been subject to validation by members of the PL/I Language Department of IBM UK Laboratories Hursley, England. The results of the checking effort conducted in Hursley have been taken into account in this present corrected form.

The formal definition given here includes more details than are given in the Specifications. These details have been confirmed as far as possible by the PL/I Language Department Hursley during the validation process. Some amendments and clarifications to the Specifications were generated during this process and will be published as Technical News Letters to the Specifications.

Contribution to the document:

Authors: M. Fleck 4., 7., parts of 2., 3., 5., 8., revision of the document
E. Neuhold the remaining part

Coordination of production: H. Hoja, W. Pachl
Production of concrete syntax and cross-reference index: K. F. Koch
Typing: H. Deim, W. Schatzl
Special graphics: G. Lehmayer

Validation: P. Seaman
# CONTENTS

1. **INTRODUCTION**  
   Page 1-1

2. **NOTATION, CONVENTIONS, AND BASIC DEFINITION TOOLS**  
   2-1
   - 2.1 Basic Functions  
   - 2.2 Some Frequently Used Predicates over Elementary Objects  
   - 2.3 Referencing

3. **CONCRETE SYNTAX**  
   3-1
   - 3.1 Generation of a Concrete Compile Time Program
     - 3.1.1 The normal generation process
     - 3.1.2 Auxiliary rules
       - 3.1.2.1 Keyword abbreviations
       - 3.1.2.2 Rules for the 48 character set version
     - 3.1.3 Production Rules
       - 3.2.1 Higher level production rules
       - 3.2.2 Lower level production rules
       - 3.2.3 List of ct-words
       - 3.2.4 Cross reference list

4. **ABSTRACT REPRESENTATION OF CONCRETE CT-PROGRAMS**  
   4-1
   - 4.1 Correspondence between a Concrete Ct-Program in the 60-character Set Version and its Abstract Representation
   - 4.2 Correspondence between a Concrete Ct-Program in the 48-character Set Version and its Abstract Representation
   - 4.3 Keyword Abbreviations
   - 4.4 Abstract Representation of Concrete Syntax

5. **THE TRANSLATOR**  
   5-1
   - 5.1 Construction of the Declaration-Part
   - 5.2 Construction of the Text-Part-List
6. ABSTRACT SYNTAX

7. INTRODUCTION TO THE INTERPRETATION OF ABSTRACT COMPILE TIME PROGRAMS

7.1 Introduction

7.1.1 Abstract compile time program

7.1.2 General concepts of the interpretation

7.2 The Declaration-Part

7.2.1 Variables

7.2.2 Labels

7.2.3 Entry-names

7.2.3.1 Structure of the associated attribute

7.2.3.2 The procedure-body

7.3 Dynamic Properties of Identifiers and their Influence on
the State

7.3.1 Scope of identifiers

7.3.2 Denotation of identifiers

7.3.3 The environment and the denotation directory

7.3.4 Representation

7.4 Interpretation

7.4.1 Interpretation of the declaration-part

7.4.2 Interpretation of the text-part-list

7.4.2.1 The state components "text storage", "statement counter" and "control dump"

7.4.2.2 Interpretation of the include statement

7.4.2.3 Interpretation of the goto statement

7.4.2.4 Interpretation of the group

7.4.2.5 Interpretation of the if statement

7.4.2.6 Interpretation of the declare statement, the activate-, and the deactivate statement

7.4.2.7 Interpretation of the assignment statement

7.4.2.8 Evaluation of expressions

7.4.3 Evaluation of references appearing in expressions

7.4.3.1 Reference of a variable

7.4.3.2 Reference of a function procedure

7.4.3.3 Reference of the built-in function SUBSTR

7.4.4 The scan and replacement mechanism for the ct-text
7.5 Summary of the State Components and their Properties
7.5.1 The external text storage ctET
7.5.2 The result cell ctR
7.5.3 The unique name counter ctUN
7.5.4 The environment ctE
7.5.5 The denotation directory ctDN
7.5.6 The procedure-body directory ctP
7.5.7 The text storage ctT
7.5.8 The statement counter ctSC
7.5.9 The control ctC
7.5.10 The control dump ctCD
7.5.11 The return information ctRI
7.5.12 The dump ctD

8. INTERPRETER
8.1 Abstract Syntax of the CT-Machine States
8.2 The Initial State of the CT-Machine
8.3 Formal Definition of the CT-Facilities Semantics
8.3.1 Program execution and interpretation of the declaration-part
8.3.2 Interpretation of a (procedure-) text-part-list
8.3.3 Interpretation of the group
8.3.4 Assignment statement, evaluation of expressions, and conversions
8.3.5 Evaluation of references appearing in ct-expressions
8.3.6 Goto statement
8.3.7 If statement
8.3.8 Activate-, deactivate-, and declare statement
8.3.9 Include statement
8.3.10 Return statement
8.3.11 Interpretation of the procedure-body
8.3.12 The scan and replacement mechanism for the ct-text

APPENDIX I : CROSS-REFERENCE INDEX

APPENDIX II : LIST OF ELEMENTARY OBJECTS
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.4</td>
<td>The additional paragraph about the main topic.</td>
</tr>
<tr>
<td>2.2.2</td>
<td>The enclosure contains another paragraph.</td>
</tr>
<tr>
<td>3.3.3</td>
<td>The paragraph with additional content.</td>
</tr>
<tr>
<td>4.4.4</td>
<td>The paragraph with more content.</td>
</tr>
<tr>
<td>5.5.5</td>
<td>The paragraph with even more content.</td>
</tr>
<tr>
<td>6.6.6</td>
<td>The paragraph with the most content.</td>
</tr>
</tbody>
</table>

**Introduction**

- 1.2.1: General outline of the introduction and its objectives.
- 2.3.3: Detailed elaboration of the introduction and its purposes.

**Chapter 1**

- 1.1.1: Overview of the chapter and its main points.
- 1.2.2: Detailed discussion of the chapter.

**Chapter 2**

- 2.1.1: Introduction to the chapter and its main points.
- 2.2.2: Detailed discussion of the chapter.

**Appendix I: List of References**

- A1.1: General reference list.
- A1.2: List of specific references.
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" /1/ and from the Formal Definition of PL/I /3,4,6/. 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 /3,4,6/ 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 a result storage (see chapters 7,8).
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 /6/ 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 ct-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 4 by a set of predicate definitions, called "abstract representation of the concrete syntax". (No constructive algorithm is given for the "parsing function" ct-parse. A special algorithm would lead to a loss of universality without contributing any information with regard to the formal definition).

4) In chapter 5 the function ct-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, the test for multiple declarations, and the testing and structuring of the attributes. 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 6 by a set of predicate definitions, called the "abstract syntax". (Although the specification of the abstract syntax is redundant - because it is given implicitly by the function ct-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 8 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 transition function. 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 concept and idea of the abstract machine is taken from the PL/I Interpreter /1, 6/.

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. In the present paper, more stress has been given to a clear syntactic structure rather than to being restrictive at the syntactic level. A certain class of syntactically correct programs are discarded by the Translator. However, in order to make Abstract Syntax together with the Interpreter 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 \( T \rightarrow \text{error} \) (in function definitions) or \( T \rightarrow \text{error} \) (in instruction definitions) for the invalid cases.

A computation of an abstract ct-program is successfully terminated only if both the ct-control and the ct-dump component of the last entered machine state contain the special "object" \( \Omega \). After a successfully terminated computation the ct-result component of the CT-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.
The space needed to perform a partitioning of the definition increases as

...
2. NOTATION, CONVENTIONS, AND BASIC DEFINITION TOOLS

Throughout the present document the notation and conventions introduced in the following sections of /3/, /4/ and /6/ are used without any special reference:
section 2.1 and 2.2 of /3/,
section 1.1 to 1.4 of /4/ (especially the selector-relations \( p \rightarrow q \) and \( p < q \) and the functions length and collect),
chapter 1 and section 12.1 of /6/.

2.1 Basic Functions

This section defines basic functions which are used for the formal definition of the compile time facilities. Most of them are handled as primitive and are not defined exactly, because their meaning appears to be evident or depends on specific implementations.

(1) \( \text{ct-af}(x,y) \)

The function \( \text{ct-af}(x,y) \) has as first argument objects \( x \) satisfying the predicates is-id or is-ad or is-*. The second argument is characterized by the predicates is-id or is-intg-val. The values are objects satisfying the predicate is-ad. It has the essential property

\[
\text{ct-af}(x,y) = \text{ct-af}(a,b) \Leftrightarrow x = a \land y = b
\]

This function defines for abstract ct-identifiers unique addresses under which their denotation can be found in the abstract CT-Machine. For this purpose the first argument holds scope information for the name given as second argument. The range of this function and the set of unique names have no elements in common. Any further specification on the values of \( \text{ct-af}(x,y) \), except their distinctness need not be given.

(2) \( \text{ct-collat}(x) \)

This function has an argument satisfying the predicate is-char-val. The values are mathematical integer values (characterized by is-intg-val) which specify the position of the argument in an implementation defined collating sequence. These values are then used in comparison operations by the CT-Interpreter.
(3) ct-truncate(t,n)

Both argument positions of this function are objects characterized by the predicate is-intg-val. The first one represents a mathematical integer value which has to be truncated depending on the implementation dependent precision n given as second argument. The function value again is a mathematical integer value, but since the function itself is implementation dependent no further specification can be given. The function is used during the interpretation of an abstract ct-program in arithmetic operations, character-to-integer-conversions, and bit-to-integer-conversions whenever the result is greater or equal $10^n$.

(4) n

The object n, a mathematical integer value, represents the implementation dependent precision of ct-integer constants.

(5) ct-represent(da,v)

The function ct-represent(da,v) transforms an abstract integer, bit or character value v into its internal value representation according to the type specification found in the first argument position. The range of arguments of this one-to-one mapping is:

$$\begin{align*}
\text{is-INTG}(da) & \land \text{is-intg-val}(v) \\
\text{is-BIT}(da) & \land v \in \{0\text{-CHAR}, 1\text{-CHAR}\} \\
\text{is-CHAR}(da) & \land \text{is-char-val}(v)
\end{align*}$$

Since the internal value representation is implementation dependent, the function is not further specified.

(6) ct-value(da, vr)

The function ct-value(da, vr) transforms an internal value representation vr into an abstract integer, bit or character value, depending on the type information (INTG, CHAR or BIT) given as first argument.

The range of arguments of this function is:

$$\{<da, vr> \mid (\exists x) (\text{ct-represent}(da,x) = vr \land (\text{is-INTG}(da) \supset \text{abs}(x) < 10^n))\}$$
Between ct-value and ct-represent the following conditions hold:

\[
\text{ct-represent}(da, \text{ct-value}(da, vr)) = vr
\]
if \(<da, vr>\) is in the range of arguments of ct-value

\[
\text{ct-value}(da, \text{ct-represent}(da, v)) =
\]
\[
is-\text{INTG}(da) \Rightarrow \text{abs}(v) < 10^n \Rightarrow v
\]
\[
\text{T} \rightarrow \text{error}
\]
if \(<da, v>\) is in the range of arguments of ct-represent.

(7) ct-sel(idp)

This function maps an object described by is-ct-id-pair onto a selector. The following condition must hold:

\[
x \neq y \Rightarrow \text{ct-sel}(x) \neq \text{ct-sel}(y).
\]

(8) mk-ct-id(x) = mk-id\text{lin-3}(x)

for: \text{is-c-ct-identifier}(x)

Ref.: lin-3 4-3(6)

(9) mk-id(cl)

This function maps special lists of character values into "abstract identifiers" (is-id). The range of arguments of this function is:

\[
\{cl | \text{is-list}(cl) \& \text{is-letter-elem}(1, cl) \&

(\forall i)[(2 \leq i \leq \text{length}(cl) \Rightarrow \text{is-alpham-char-elem}(i, cl)]
\}

The following condition holds:

\[
x \neq y \Rightarrow \text{mk-id}(x) \neq \text{mk-id}(y)
\]
for each \(x\) and \(y\) out of the range of arguments of mk-id.

In some applications of the function mk-id the following shorthand notation is used:

mk-id(SUBSTR) stands for mk-id(<S-CHAR,U-CHAR,B-CHAR,S-CHAR,T-CHAR,R-CHAR>).
(10) headlist(x) =
    is-list(x) & \neg is-<>(x) \rightarrow
    \neg (length(x) = 1 \rightarrow <>),
    \neg \neg (length(x) = 1 \rightarrow <>,
    \neg \neg (length(x) = 1 \rightarrow <>,
    T \rightarrow error

(11) last-el(x) =
    is-<>(x) \rightarrow Q
    is-list(x) \rightarrow elem(length(x), x)
    T \rightarrow error

2.2 Some frequently used Predicates over Elementary Objects

All following classes of elementary objects are mutually disjoint.

(12) is-intg-val

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

(13) is-ct-vr

This predicate characterizes the range of values of the function ct-represent (cf. 2.1).

(14) is-ad

This predicate characterizes the range of values of the function ct-ad (cf. 2.1).

(15) is-n

This predicate characterizes the enumerably infinite set of unique names \( n_0, n_1, n_2, \ldots \).

(16) is-id

This predicate characterizes the range of values of the function mk-id (cf. 2.1).
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:

function-name chapter - page (formula-number)

There are the following exceptions to this rule:

1) All names defined in places listed in the headnote of chapter 2 are not referenced.

2) All names defined in section 2.2 are not referenced.

3) All names defined in the abstract representation of concrete syntax (section 4.4, predicate names starting with is-c-...) and abstract syntax (chapter 6) are not referenced.

4) All names defined in the same section are not referenced.
Reference 11.1 Function and Instruction name occurring in a function and instruction may be referenced by a reference to the following notation:

\[ \text{Reference - Name} \]

The following reference types are permitted:

1. Function Name - Chapter - Page (family-name)

There are the following exceptions to this rule:

- All names defined in the header of the reference are referenced.
- All names defined in section 1.1 are not referenced.
- All names defined in section 1.2 are not referenced.
- All names defined in section 1.3 and subsequent sections are not referenced.
- All names occurring in the same section are not referenced.
3. CONCRETE SYNTAX

This chapter contains the syntactic description of concrete PL/I compile time programs (source text), 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 Compile Time 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 three steps:

1) Starting with the notation variable "ct-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 22 ct-words

\[ = + - * / ( ) , . ; & | \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \leftarrow \rightarrow \left arrow
With the production rule

\[
\text{ct-space ::= \{\text{blank | ct-comment}\}***}
\]

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".

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 Auxiliary rules

The compile time facilities contain the possibility of using keyword-abbreviations and also to use a 48 character set. These facilities are not given in section 3.2 because they would lengthen unnecessarily the production rules.

3.1.2.1 Keyword abbreviations

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

- ACTIVATE
- CHARACTER
- DEACTIVATE
- DECLARE
- PROCEDURE

The following abbreviations can be used instead of the corresponding keywords. The necessary replacement must be done before step 3 of the generation process takes place.
3.1.2.2 Rules for the 48 character set version

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 ct-delimiters and the other resulting ct-words as ct-non-delimiters.

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

ct-letter
ct-alphameric-character
c-t-string-character
c-t-text-character
c-t-string-part-char
c-t-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 "ct-identifier" must finally be replaced by any of them.
4) The lower level production rule for et-text has to be replaced by

```plaintext
cr-text ::=  
  { cr-space | / } { ct-text-character | ct-string-part }  
where from the production rule for ct-text-character the character 
"blank" is deleted.
```

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

3.2.1 Higher level production rules

(1) ct-program ::= 
ct-text | ct-program ct-sentence ct-text

(2) ct-sentence ::= 
ct-statement | ct-procedure | ct-declare-statement

(3) ct-statement ::= 
ct-if-statement | ct-unconditional-statement

(4) ct-if-statement ::= 
ct-if-clause ct-statement | 
ct-if-clause ct-balanced-statement % ELSE ct-statement

(5) ct-if-clause ::= 
% [ ct-labellist ] IF ct-expression % THEN

(6) ct-labellist ::= 
[ ct-identifier : ]***

(7) ct-balanced-statement ::= 
ct-if-clause ct-balanced-statement % ELSE 
ct-balanced-statement | ct-unconditional-statement

(8) ct-unconditional-statement ::= 
% [ ct-labellist ] ct-proper-statement

(9) ct-proper-statement ::= 
ct-do-group | ct-assignment-statement | ct-goto-statement | 
ct-null-statement | ct-activate-statement | 
ct-deactivate-statement | ct-include-statement
(10) \( \text{ct-do-group} ::= \)
    \( \text{ct-simple-group} \mid \text{ct-iterated-group} \)

(11) \( \text{ct-simple-group} ::= \)
    \( \text{DO} \ ; \text{ct-program} \text{ct-end-clause} \)

(12) \( \text{ct-end-clause} ::= \)
    \( \% [ \text{ct-labellist} ] \text{END} [ \text{ct-identifier} ] ; \)

(13) \( \text{ct-iterated-group} ::= \)
    \( \text{DO} \text{ct-do-specification} \ ; \text{ct-program} \text{ct-end-clause} \)

(14) \( \text{ct-do-specification} ::= \)
    \( \text{ct-simple-reference} = \text{ct-expression} \mid \text{BY} \text{ct-expression} \)
    \( \mid [ \text{TO} \text{ct-expression} ] \mid \text{TO} \text{ct-expression} \mid \text{BY} \text{ct-expression} \)

(15) \( \text{ct-simple-reference} ::= \)
    \( \text{ct-identifier} \)

(16) \( \text{ct-assignment-statement} ::= \)
    \( \text{ct-simple-reference} = \text{ct-expression} ; \)

(17) \( \text{ct-expression} ::= \)
    \( \text{ct-expression-six} \mid \text{ct-expression} \mid \text{ct-expression-six} \)

(18) \( \text{ct-expression-six} ::= \)
    \( \text{ct-expression-five} \mid \text{ct-expression-six} \& \text{ct-expression-five} \)

(19) \( \text{ct-expression-five} ::= \)
    \( \text{ct-expression-four} \mid \text{ct-expression-five} \text{ct-comparison-operator} \text{ct-expression-four} \)

(20) \( \text{ct-comparison-operator} ::= \)
    \( > \ | \ >= \ | \ = \ | \ <= \ | \ < \ | \ -> \ | \ -= \ | \ <- \)

(21) \( \text{ct-expression-four} ::= \)
    \( \text{ct-expression-three} \mid \text{ct-expression-four} \ll \text{ct-expression-three} \)

(22) \( \text{ct-expression-three} ::= \)
    \( \text{ct-expression-two} \mid \text{ct-expression-three} \{ \pm \} \text{ct-expression-two} \)

3.2.1
(23) ct-expression-two ::= ct-expression-one | ct-expression-two [ * | / ] ct-expression-one

(24) ct-expression-one ::= ct-primitive-expression | [ + | - | - ] ct-expression-one

(25) ct-primitive-expression ::= ( ct-expression ) | ct-reference | ct-constant

(26) ct-reference ::= ct-identifier [ ( ct-expression ) ]

(27) ct-goto-statement ::= { SOSTO | SOT TO } ct-identifier ;

(28) ct-null-statement ::= ;

(29) ct-return-statement ::= RETURN ( ct-expression ) ;

(30) ct-activate-statement ::= ACTIVATE [ , ct-identifier ] ;

(31) ct-deactivate-statement ::= DEACTIVATE [ , ct-identifier ] ;

(32) ct-include-statement ::= INCLUDE [ , ct-library-specification ] ;

(33) ct-library-specification ::= [ ct-identifier ] ct-identifier ct-identifier

(34) ct-procedure ::= % ct-entry-namelist PROCEDURE [ ct-parameterlist ] [ CHARACTER | FIXED ] ; [ ct-p-sentencelist ] ct-end-clause

(35) ct-entry-namelist ::= [ ct-identifier ]
(36) \text{ct-parameterlist ::=}
\quad ( \{ , * \text{ct-identifier} \} )

(37) \text{ct-p-sentencelist ::=}
\quad \text{ct-p-sentence} \ldots

(38) \text{ct-p-sentence ::=}
\quad \text{ct-p-statement} | \text{ct-p-declare-statement}

(39) \text{ct-p-statement ::=}
\quad \text{ct-p-if-statement} | \text{ct-p-unconditional-statement}

(40) \text{ct-p-if-statement ::=}
\quad \text{ct-p-if-clause} \text{ct-p-statement} | \text{ct-p-if-clause} \text{ct-p-balanced-statement} \text{ELSE} \text{ct-p-statement}

(41) \text{ct-p-if-clause ::=}
\quad [ \text{ct-labellist} ] \text{IF} \text{ct-expression} \text{THEN}

(42) \text{ct-p-balanced-statement ::=}
\quad \text{ct-p-if-clause} \text{ct-p-balanced-statement} \text{ELSE} \text{ct-p-balanced-statement} | \text{ct-p-unconditional-statement}

(43) \text{ct-p-unconditional-statement ::=}
\quad [ \text{ct-labellist} ] \text{ct-p-proper-statement}

(44) \text{ct-p-proper-statement ::=}
\quad \text{ct-p-do-group} | \text{ct-assignment-statement} | \text{ct-goto-statement} | \text{ct-null-statement} | \text{ct-return-statement}

(45) \text{ct-p-do-group ::=}
\quad \text{ct-p-simple-group} | \text{ct-p-iterated-group}

(46) \text{ct-p-simple-group ::=}
\quad \text{DO ; [ ct-p-sentencelist ] ct-p-end-clause}

(47) \text{ct-p-end-clause ::=}
\quad [ \text{ct-labellist} ] \text{END [ ct-identifier ] ;}

3.2.1
(48) \text{ct-p-iterated-group} ::= \\
\hspace{1cm} \text{DO ct-do-specification } ; \begin{array}{c}
\text{ct-p-sentencelist } \text{ct-p-end-clause}
\end{array}

(49) \text{ct-p-declare-statement} ::= \\
\hspace{1cm} \begin{array}{c}
\text{[ [ ct-identifier : ] } \cdots \text{ ] DECLARE ct-p-declarationlist ;}
\end{array}

(50) \text{ct-p-declarationlist} ::= \\
\hspace{1cm} \begin{array}{c}
\{ , \text{ ct-p-declaration} \cdots \}
\end{array}

(51) \text{ct-p-declaration} ::= \\
\hspace{1cm} \begin{array}{c}
\text{ct-identifier } \begin{array}{c}
\text{ct-p-declarationlist } \}
\end{array} \begin{array}{c}
\text{ct-p-attribute} \cdots \}
\end{array}
\end{array}

(52) \text{ct-p-attribute} ::= \\
\hspace{1cm} \begin{array}{c}
\text{FIXED | CHARACTER}
\end{array}

(53) \text{ct-declare-statement} ::= \\
\hspace{1cm} \begin{array}{c}
\% \text{[ [ ct-identifier : ] } \cdots \text{ ] DECLARE ct-declarationlist ;}
\end{array}

(54) \text{ct-declarationlist} ::= \\
\hspace{1cm} \begin{array}{c}
\{ , \text{ ct-declaration} \cdots \}
\end{array}

(55) \text{ct-declaration} ::= \\
\hspace{1cm} \begin{array}{c}
\text{ct-identifier } \begin{array}{c}
\text{ct-declarationlist } \}
\end{array} \begin{array}{c}
\text{ct-attribute} \cdots \}
\end{array}
\end{array}

(56) \text{ct-attribute} ::= \\
\hspace{1cm} \begin{array}{c}
\text{FIXED | CHARACTER | ct-entry-attribute | ct-returns-attribute}
\end{array}

(57) \text{ct-entry-attribute} ::= \\
\hspace{1cm} \begin{array}{c}
\text{ENTRY} \begin{array}{c}
\text{ct-descriptorlist } \}
\end{array}
\end{array}

(58) \text{ct-descriptorlist} ::= \\
\hspace{1cm} \begin{array}{c}
\text{ct-descriptor } \begin{array}{c}
\text{ct-descriptorlist } \}
\end{array}
\end{array}

(59) \text{ct-descriptor} ::= \\
\hspace{1cm} \begin{array}{c}
\text{FIXED | CHARACTER}
\end{array}

(60) \text{ct-returns-attribute} ::= \\
\hspace{1cm} \begin{array}{c}
\text{RETURNS } \begin{array}{c}
\text{ct-descriptorlist } \}
\end{array}
\end{array}

3.2.1
### 3.2.2 Lower level production rules

(61) ct-identifier ::= 
     ct-letter [ ct-alphameric-character ]

(62) ct-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 | $ | @ | #

(63) ct-alphameric-character ::= 
     ct-letter | ct-digit | _

(64) ct-digit ::= 
     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(65) ct-constant ::= 
     ct-integer | ct-character-string | ct-bit-string

(66) ct-integer ::= 
     [ ct-integer ] ct-digit

(67) ct-bit-string ::= 
     ' [ ct-bit-string ] ' B

(68) ct-bit ::= 
     0 | 1

(69) ct-character-string ::= 
     ' [ ct-string-character ] '

(70) ct-string-character ::= 
     ct-alphameric-character | blank | 
     ' ' | = | + | - | * | / | ( | ) | , | । | ; | : | & | | ~ | > | < | ? | % | extralingual-character

(71) ct-text ::= 
     [ *** ] | /*** | { [ *** | /*** ] [ ct-text-character | ct-string-part | ct-comment ] }*** [ *** | /*** ]
(72) ct-text-character ::= 
    ct-alphameric-character | blank | = | + | - | ( | ) | , | |
                     - | ; | : | & | | | | | | > | | | | |
(73) ct-string-part ::= 
    ' [ ct-string-part-char*** ] ' 
(74) ct-string-part-char ::= 
    ct-alphameric-character | blank | 
    = | + | - | * | / | ( | ) | , | - | ; | : | |
    & | | | | | | > | | | | | | < | | | | |
    | | | | | | | | extralingual-character 
(75) ct-comment ::= 
    /* [ [ ** ** ] ct-comment-symbol | / ]*** ] **** */ 
(76) ct-comment-symbol ::= 
    ct-alphameric-character | blank | 
    = | + | - | ( | ) | , | - | ; | : | | | & | |
    | | | | | | > | | | | | | < | | | | |
    | | | | | | | | extralingual-character
3.2.3 List of ct-words

<
<=
(
+
-
*
/
%
>
>=
:

ACTIVATE
BY
CHARACTER
cf-constant
cf-identifier
cf-text
DEACTIVATE
DECLARE
DO
ELSE
END
ENTRY
FIXED
GO
GOTO
IF
INCLUDE
PROCEDURE
RETURN
RETURNS
THEN
TO
3.2.4 Cross reference list

3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
< 3-6 (20), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
<= 3-6 (20)
{ 3-7 (25), 3-7 (26), 3-7 (29), 3-7 (33), 3-8 (36), 3-9 (51), 3-9 (55), 3-9 (57), 3-9 (60), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
+ 3-6 (22), 3-7 (24), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
\ 3-6 (17), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
| 3-6 (21)
\ 3-6 (18), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
\ 3-10 (62)
* 3-7 (23), 3-10 (70), 3-10 (71), 3-10 (71), 3-10 (71), 3-11 (74), 3-11 (75), 3-11 (75), 3-11 (75)
} 3-7 (25), 3-7 (26), 3-7 (29), 3-7 (33), 3-8 (36), 3-9 (51), 3-9 (55), 3-9 (57), 3-9 (60), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
; 3-6 (11), 3-6 (12), 3-6 (13), 3-6 (16), 3-7 (27), 3-7 (29), 3-7 (29), 3-7 (30), 3-7 (31), 3-7 (32), 3-7 (34), 3-8 (46), 3-8 (47), 3-9 (48), 3-9 (53), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
; 3-7 (24), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
< 3-6 (20)
=> 3-6 (20)
< 3-6 (20)
- 3-6 (22), 3-7 (24), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
/ 3-7 (23), 3-10 (70), 3-10 (71), 3-10 (71), 3-10 (71), 3-11 (74), 3-11 (75), 3-11 (75), 3-11 (75)
, 3-7 (26), 3-7 (30), 3-7 (31), 3-7 (33), 3-8 (36), 3-9 (52), 3-9 (54), 3-9 (58), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
% 3-5 (4), 3-5 (5), 3-5 (5), 3-5 (7), 3-5 (9), 3-6 (12), 3-7 (34), 3-9 (53), 3-10 (70), 3-11 (74), 3-11 (76)
> 3-6 (20), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
>= 3-6 (20)

3.2.4
3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
3-5 (6), 3-7 (35), 3-9 (49), 3-9 (53), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
3-10 (62)
3-10 (62)
3-10 (67), 3-10 (67), 3-10 (69), 3-10 (69), 3-11 (73), 3-11 (73), 3-11 (76)
3-10 (70)
3-6 (14), 3-6 (16), 3-6 (20), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
3-10 (62)
3-7 (30)
3-10 (62), 3-10 (67)
3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
3-6 (14), 3-6 (14)
3-10 (62)
3-7 (34), 3-9 (52), 3-9 (56), 3-9 (59), 3-9 (60)
3-5 (9), 3-7 (39)
3-10 (61), 3-10 (63), 3-10 (70), 3-11 (72), 3-11 (74), 3-11 (76)
3-5 (9), 3-6 (16), 3-8 (44)
3-9 (55), 3-15 (56)
3-5 (4), 3-5 (7), 3-5 (72), 3-5 (7)
3-10 (67), 3-10 (68)
3-10 (65), 3-10 (67)
3-10 (65), 3-10 (69)
3-10 (71), 3-11 (25)
3-11 (75), 3-11 (26)
3-6 (19), 3-6 (20)
3-7 (25), 3-9 (165)
3-5 (9), 3-7 (31)
3-9 (54), 3-9 (55)
3-9 (53), 3-9 (54), 3-9 (55)
3-5 (2), 3-9 (53)
ct-descriptor
ct-descriptorlist
ct-digit
ct-do-group
ct-do-specification
ct-end-clause
ct-entry-attribute
ct-entry-namelist
ct-expression
ct-expression-five
ct-expression-four
ct-expression-one
ct-expression-six
ct-expression-three
ct-expression-two
ct-goto-statement
ct-identifier
ct-if-clause
ct-if-statement
ct-include-statement
ct-integer
ct-iterated-group
cr-label-list
ct-letter
ct-library-specification
ct-null-statement
ct-p-attribute
ct-p-balanced-statement
ct-p-declaration ........................................ 3-9(53), 3-9(51)
ct-p-declarationlist ................................... 3-9(49), 3-9(52), 3-9(51)
ct-p-declare-statement .................................. 3-8(38), 3-9(51)
ct-p-do-group ........................................... 3-8(44), 3-8(51)
ct-p-end-clause ......................................... 3-8(46), 3-8(47), 3-9(48)
ct-p-if-clause ........................................... 3-8(40), 3-8(40), 3-8(41), 3-8(42)
ct-p-if-statement ......................................... 3-8(39), 3-8(40)
ct-p-iterate1-group .................................... 3-8(45), 3-8(49)
ct-p-proper-statement ................................... 3-9(43), 3-9(44)
ct-p-sentence ............................................ 3-9(37), 3-8(38)
ct-p-sentencelist ....................................... 3-7(34), 3-8(37), 3-8(46), 3-9(48)
ct-p-simple-group ....................................... 3-8(45), 3-8(49)
ct-p-statement .......................................... 3-9(38), 3-8(39), 3-8(40), 3-8(40)
ct-p-unconditional-statement ......................... 3-8(39), 3-8(42), 3-8(43)
ct-parameterlist ......................................... 3-7(34), 3-8(36)
ct-primitive-expression ................................ 3-7(24), 3-7(25)
ct-procedure .............................................. 3-5(21), 3-7(34)
ct-program ................................................. 3-5(11), 3-5(1), 3-6(11), 3-6(13)
ct-proper-statement ..................................... 3-5(8), 3-5(91)
ct-reference .............................................. 3-7(25), 3-7(30)
ct-return-statement ...................................... 3-7(29), 3-8(44)
ct-returns-attribute ..................................... 3-9(56), 3-9(60)
ct-sentence ............................................... 3-5(1), 3-5(2)
ct-simple-group ......................................... 3-6(19), 3-6(11)
ct-simple-reference ..................................... 3-6(14), 3-6(15), 3-6(16)
ct-string-character ...................................... 3-10(69), 3-10(70)
ct-string-part ........................................... 3-10(71), 3-11(73)
ct-string-part-char ..................................... 3-11(73), 3-11(74)
ct-text ...................................................... 3-5(1), 3-5(1), 3-10(71)
RETURN

RETURNS

THEN

TO

V

W

X

Y

Z

0

1

2

3

4

5

6

7

8

9

3-7 (29)

3-9 (60)

3-10 (62)

3-5 (5), 3-8 (41)

3-6 (14), 3-7 (27)

3-10 (62)

3-10 (62)

3-10 (62)

3-10 (62)

3-10 (62)

3-10 (64), 3-10 (68)

3-10 (64), 3-10 (68)

3-10 (64)

3-10 (64)

3-10 (64)

3-10 (64)

3-10 (64)

3-10 (64)

3-10 (64)
4. ABSTRACT REPRESENTATION OF CONCRETE CT-PROGRAMS

The "abstract representation" of a concrete ct-program is a structured object (similar to the parsing tree of that concrete ct-program) which satisfies the predicate is-c-ct-program. This predicate is defined in 4.4 by a set of predicate definitions, called "abstract representation of the concrete syntax".

The abstract representation of the concrete syntax may be considered also as a normal form with regard to the two different production systems, corresponding to the 60- and to the 48-character-set versions. 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 an elementary object characterized by the predicate is-GT, independently from the concrete representation.

Each predicate definition out of the abstract representation of the concrete syntax is closely related to a production rule out of the 60-character set production system. This one-to-one mapping is described in /4/, AI. (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-ct-program-48, such that is-c-ct-program-48 ⇒ is-c-ct-program).

In the following two parsing functions are defined, namely ct-parse and ct-parse-48, mapping a concrete ct-program of the 60- and of the 48-character set, respectively, onto its abstract representation. For this purpose, the concrete ct-program which is a string of concrete PL/I characters, is represented by a list of abstract character values, i.e. of elementary objects satisfying the predicate is-char-val (cf. 6-4 (46)). The correspondence between the individual concrete characters and these character values is given in /4/, AI.

Metavariables

<table>
<thead>
<tr>
<th>text</th>
<th>is-char-val-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>is-c-ct-program</td>
</tr>
<tr>
<td>x</td>
<td>any components of t or a list of such components</td>
</tr>
<tr>
<td></td>
<td>a concrete ct-program</td>
</tr>
<tr>
<td></td>
<td>abstract representation of a concrete ct-program</td>
</tr>
</tbody>
</table>
4.1 Correspondence between a Concrete Ct-programm in the 60-character Set Version and its Abstract Representation

(1) \(\text{ct-parse}(\text{text}) = (t) (\text{text} \in \text{linearize}(t) \& \text{is-c-ct-program}(t))\)

(2) \(
\text{linearize}(t) = \\
\{ \text{lin-3}(x) \mid x \in \text{insert-space} \cdot \text{lin-1}(t) \}
\)

(3) \(\text{lin-1}(t) = \\
\text{is-Q}(t) < \alpha > \\
\text{is-c-ct-text}(t) < \alpha > \text{lin-2}(t) < \alpha > \\
\text{T} < \text{lin-1} \cdot \text{s_1}(t) > < \text{lin-2} \cdot \text{s_2}(t) > < \text{lin-1} \cdot \text{s_3}(t) >
\)

Note: The function \(\text{lin-1}\) linearizes the main structure of \(t\) according to the first predicate definition of 4.4 into a list, thereby inserting between a ct-sentence and a ct-text or between two ct-sentences the dummy symbol "\(\alpha\)" indicating where the insertion of ct-space is forbidden.

(4) \(\text{lin-2}(x) = \\
\text{is-Q}(x) < > \\
\text{s-length}(x) = 0 < x > \\
\text{is-c-ct-simple-group}(x) < s_1(x) > < s_2(x) > < \text{lin-1} \cdot \text{s_3}(x) > < \text{lin-2} \cdot s_4(x) > \\
\text{is-c-ct-iterated-group}(x) < s_1(x) > < \text{lin-2} \cdot s_2(x) > < s_3(x) > < \text{lin-1} \cdot s_4(x) > < \text{lin-2} \cdot s_5(x) > \\
\text{T} < \text{s-length}(x) > \text{lin-2} \cdot \text{s_1}(x) < \text{CDNC} \ (\text{lin-2} \cdot \text{s-del}(x) < \text{lin-2} \cdot s_1(x)) >
\)

Note: The function \(\text{lin-2}\) planes \(x\) into a list of those components which are syntactical units that may not be interrupted by space (blanks and comments). This is done by planing the structure given by selectors of the form \(s_i\) or \(s\)-del, but not affecting the structure given by selectors of the form \(\text{elem}(i)\). Note that do-groups outside procedures contain an abstract represented ct-program, hence the function \(\text{lin-1}\) must be applied again.
(5) \( \text{insert-space}(x) = \)

\[
\begin{align*}
\text{is-<>}(x) & \rightarrow \{< >\} \\
\text{head}(x) & = \alpha \rightarrow \text{insert-space}\cdot\text{tail}(x) \\
\text{head}\cdot\text{tail}(x) & = \alpha \\
\{<\text{head}(x)> \rightarrow \gamma \mid \gamma \in \text{insert-space}\cdot\text{tail}\cdot\text{tail}(x)\} \\
\text{is-c-ct-delimiter}\cdot\text{head}(x) & \rightarrow \text{is-c-ct-delimiter}\cdot\text{head}\cdot\text{tail}(x) \\
\{\text{mk-list}(\text{head}(x),y,z) \mid (\text{is-c-ct-space}(y) \lor \text{is-}\& \gamma) \land \gamma \in \text{insert-space}\cdot\text{tail}(x)\} \\
\end{align*}
\]

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 interspersing satisfying the condition that at least between all pairs of consecutive ct-non-delimiters spaces are inserted, unless the symbol "\(\alpha\)" occurs, which has to be omitted but not replaced by spaces.

(6) \( \text{lin-3}(x) = \)

\[
\begin{align*}
\text{is-\&}(x) & \rightarrow <> \\
\text{is-char-val}(x) & \rightarrow <x> \\
\text{lg}_x & \rightarrow \text{CONC} \cdot \text{lin-3}\cdot\text{elem}(i,x) \\
T & \rightarrow \text{CONC} \cdot \text{lin-3}\cdot\text{elem}(i,x) \\
\end{align*}
\]

where: \( \text{lg}_x = (i) \neg\text{is-\&}\cdot\text{elem}(i,x) \land (\forall j)(j > i \Rightarrow \neg\text{is-\&}\cdot\text{elem}(j,x)) \)

Note: The function lin-3 linearizes lists whose elements are either elementary objects or again such lists into a list of elementary objects.

(7) \( \text{mk-list}(a,b,\text{list}) = \)

\[
\mu\langle<\text{elem}(1):a>,<\text{elem}(2):b>\rangle\uparrow \text{list}
\]

for: \( \neg\text{is-\&}(a), \text{is-list}(\text{list}) \)

(8) \( \text{is-c-ct-delimiter}(x) = \)

\[
x \in \{\text{EQ, PLUS, MINUS, ASTER, SLASH, LEFT-PAR, RIGHT-PAR, COMMA, POINT, SEMIC, COLON, AND, OR, NOT, GT, LT, OR, OR}, <\text{GT, EQ}, <\text{LT, EQ}, <\text{NOT, GT}, <\text{NOT, EQ}, <\text{NOT, LT}\}
\]
4.2 Correspondence between a Concrete Ct-program in the 48-character Set Version and its Abstract Representation.

The following functions together with the concrete syntax of the 48-character set version contain some redundancy. The reason for this is that two methods for the generation of a program are conceivable, namely either to generate a concrete program as described in chapter 3 and to map the result by means of (10) onto its abstract representation, or to generate directly the abstract representation according to the set of predicate definitions, given in 4.4. In the second case linearize-48(t) yields the set of corresponding concrete programs in the 48-character version. Hence linearize-48 has also to check whether its argument belongs to that subclass of is-c-ct-program whose members are representable as proper concrete programs in the 48-character version.

(10) \text{ct-parse-48}(text) =
\{(t) \mid \text{text} \in \text{linearize-48}(t) \& \text{is-c-ct-program}(t)\}

(11) \text{linearize-48}(t) =
\neg(\exists p)(\text{is-c-ct-identifier} \cdot p(t) \&
\text{(is-c-NOT} \cdot \text{lin-3} \cdot p(t) \text{)} \lor \text{is-c-AND} \cdot \text{lin-3} \cdot p(t) \lor
\text{is-c-OR} \cdot \text{lin-3} \cdot p(t) \lor \text{is-c-GT} \cdot \text{lin-3} \cdot p(t) \lor
\text{is-c-LT} \cdot \text{lin-3} \cdot p(t) \lor \text{is-c-GE} \cdot \text{lin-3} \cdot p(t) \lor
\text{is-c-NL} \cdot \text{lin-3} \cdot p(t) \lor \text{is-c-NE} \cdot \text{lin-3} \cdot p(t) \lor
\text{is-c-CAT} \cdot \text{lin-3} \cdot p(t) \rightarrow
\text{lin-3}(x) \land x: \text{insert-space-48} \cdot \text{replace-48} \cdot \text{lin-1-48}(t) \&
\neg(\exists i)(\text{elem}(i, \text{lin-3}(x)) \epsilon
\{\text{COMM-ACT, NUMBER-SIGN, BREAK, SEMIC, COLON, AND, OR, NOT, GT, LT, QUEST, PERC}\})\}
\rightarrow \text{error}

\text{Ref.: lin-3 4-3(6)}

Note: cf. points 1) to 3) of section 3.1.2.2
(12) \( \text{lin-1-48}(t) = \) 
\[
\begin{align*}
is-\Omega(t) & \rightarrow \langle \alpha \rangle \\
is-c\text{-ct-text}(t) & \rightarrow \\
(\sim(\exists i) \text{(is-SLASH=elem}(1,e_l) \& \text{is-SLASH=elem}(2,e_l) \& \\
(is-\text{BLANK}(e_l) \vee \text{is-c\text{-ct-comment}(e_l)))) & \rightarrow \langle \alpha \rangle \langle t \rangle \langle \alpha \rangle , \\
T & \rightarrow \text{error}
\end{align*}
\]

\[ T \rightarrow \text{lin-1-48} * s_1(t) \text{ lin-2-48} * s_2(t) \text{ lin-1-48} * s_3(t) \]

where: \( e_l = \text{elem}(1) \cdot \text{elem}(i) \cdot \text{elem}(1,t) \)
\( e_l = \text{elem}(2) \cdot \text{elem}(i) \cdot \text{elem}(1,t) \)

Note: The restriction on ct-text is necessary for reasons of unambiguity; cf. point 4) of section 3.1.2.2

(13) \( \text{lin-2-48}(x) = \) 
\[
\begin{align*}
is-\Omega(x) & \rightarrow \langle \rangle \\
\text{slength}(x) = 0 & \rightarrow \langle x \rangle \\
is-c\text{-ct-simple-group}(x) & \rightarrow \\
\langle s_1(x) \rangle \langle s_2(x) \rangle \text{ lin-1-48} * s_3(x) \text{ lin-2-48} * s_4(x) & \\
is-c\text{-ct-iterated-group}(x) & \rightarrow \\
\langle s_1(x) \rangle \text{ lin-2-48} * s_2(x) \langle s_3(x) \rangle \text{ lin-1-48} * s_4(x) \text{ lin-2-48} * s_5(x) & \\
T & \rightarrow \text{lin-2-48} * s_1(x) \text{ CONC (lin-2-48} * \text{del}(x) \text{ lin-2-48} * s_1(x))
\end{align*}
\]

(14) \( \text{replace-48}(x) = \) 
\[
\begin{align*}
\text{length}(x) & \\
\text{LIST} & \text{ replace-48} -1 \cdot \text{elem}(i,x)
\end{align*}
\]
(15) replace-48-1(x) =

\[\begin{align*}
is-\text{COLON}(x) & \rightarrow \text{POINT, POINT} \\
is-\text{SEMIC}(x) & \rightarrow \text{COMMA, POINT} \\
is-\text{PERC}(x) & \rightarrow \text{SLASH, SLASH} \\
is-\text{NOT}(x) & \rightarrow \text{N-CHAR, O-CHAR, T-CHAR} \\
is-\text{AND}(x) & \rightarrow \text{A-CHAR, N-CHAR, D-CHAR} \\
is-\text{OR}(x) & \rightarrow \text{O-CHAR, R-CHAR} \\
is-\text{GT}(x) & \rightarrow \text{G-CHAR, T-CHAR} \\
is-\text{LT}(x) & \rightarrow \text{L-CHAR, T-CHAR} \\
x = <\text{GT}, \text{EQ}, \text{EQ}> & \rightarrow \text{G-CHAR, E-CHAR} \\
x = <\text{LT}, \text{EQ}, \text{EQ}> & \rightarrow \text{L-CHAR, E-CHAR} \\
x = <\text{NOT}, \text{GT}, \text{GT}> & \rightarrow \text{N-CHAR, G-CHAR} \\
x = <\text{NOT}, \text{LT}, \text{LT}> & \rightarrow \text{N-CHAR, L-CHAR} \\
x = <\text{NOT}, \text{EQ}, \text{EQ}> & \rightarrow \text{N-CHAR, E-CHAR} \\
x = <\text{OR}, \text{OR}> & \rightarrow \text{C-CHAR, A-CHAR, T-CHAR} \\
T & \rightarrow \text{x}
\end{align*}\]

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

\[\begin{align*}
is->(x) & \rightarrow \{<\}\} \\
\text{head}(x) = \alpha & \rightarrow \text{insert-space-48}\ast\text{tail}(x) \\
\text{head}\ast\text{tail}(x) = \alpha & \rightarrow \{<\text{head}(x)> \uparrow z \mid z \in \text{insert-space-48}\ast\text{tail}\ast\text{tail}(x)\} \\
is-\text{c-ct-delim}\text{iter}-48\ast\text{head}(x) & \vee is-\text{c-ct-delim}\text{iter}-48\ast\text{head}\ast\text{tail}(x) \rightarrow \{\text{mk-list}(\text{head}(x), y, z) \mid (is-\text{c-ct-space}(y) \vee is-\emptyset(y)) \& z \in \text{insert-space-48}\ast\text{tail}(x)\} \\
T & \rightarrow \{\text{mk-list}(\text{head}(x), y, z) \mid is-\text{c-ct-space}(y) \& z \in \text{insert-space-48}\ast\text{tail}(x)\}\}
\]

Ref.: mk-list 4-3(7), is-c-ct-space 4-4(9)

(17) is-c-ct-delim\text{iter}-48(x) =

\[\begin{align*}
is-\text{c-ct-delim}\text{iter}(x) & \vee x = \text{POINT, POINT} \vee x = \text{COMMA, POINT}
\]

Ref.: is-c-ct-delim\text{iter} 4-3(8)
4.3 Keyword Abbreviations

To include the possibility of abbreviating keywords one has to replace the predicates is-c-[name], where [name] is one of the keywords listed in the following table, throughout the sections 4 and 5 by the corresponding predicates is-c-abbr-[name] defined by:

\[
\text{is-c-abbr-[name]} = \text{is-c-[name]} \lor \text{is-c-[abbr-name]}
\]

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

<table>
<thead>
<tr>
<th>[name]</th>
<th>[abbr-name]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE</td>
<td>ACT</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>CHAR</td>
</tr>
<tr>
<td>DEACTIVATE</td>
<td>DÉACT</td>
</tr>
<tr>
<td>DECLARE</td>
<td>DCL</td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>PROC</td>
</tr>
</tbody>
</table>
The abstract representation of the concrete syntax is a set of predicate definitions which together define the predicate \texttt{is-ct-program}.

Predicates of the form \texttt{is-c-[name]}, where \texttt{name} is a string of capital letters, occurring in the following are defined by the following scheme:

\[
is-c-[\text{name}] = (\langle \text{elem}(1):\text{is-char}_1\rangle, \langle \text{elem}(n):\text{is-char}_n\rangle)
\]

where: \text{char}_i (1 \leq n) characterizes uniquely the character value corresponding to the \text{i}-th member of the string forming \texttt{name}, and \text{n} (\text{n} \geq 1) is the length of the string.

Note: Example: \texttt{is-c-IF} =

\[
(\langle \text{elem}(1):\text{is-CHAR}\rangle, \langle \text{elem}(2):\text{is-F-CHAR}\rangle)
\]

(1) \texttt{is-c-ct-program} =

\[
is-c-ct-text \lor (\langle s(1):is-c-ct-program\rangle, \langle s(2):is-c-ct-sentence\rangle, \langle s(3):is-c-ct-text\rangle)
\]

(2) \texttt{is-c-ct-sentence} =

\[
is-c-ct-statement \lor is-c-ct-procedure \lor is-c-ct-declare-statement
\]

(3) \texttt{is-c-ct-statement} =

\[
is-c-ct-if-statement \lor is-c-ct-unconditional-statement
\]
(4) is-c-ct-if-statement =
   (<s(1):is-c-ct-if-clause>,
    <s(2):is-c-ct-statement>) v
   (<s(1):is-c-ct-if-clause>,
    <s(2):is-c-ct-balanced-statement>,
    <s(3):is-PERC>,
    <s(4):is-c-ELSE>,
    <s(5):is-c-ct-statement>)

(5) is-c-ct-if-clause =
   (<s(1):is-PERC>,
    <s(2):is-0 v is-c-ct-labelist>,
    <s(3):is-c-IF>,
    <s(4):is-c-expression>,
    <s(5):is-PERC>,
    <s(6):is-c-THEN>)

(6) is-c-ct-labelist =
   (<s(1):(<s(1):is-c-ct-identifier>,
     <s(2):is-COLON>)>,...)

(7) is-c-ct-balanced-statement =
   (<s(1):is-c-ct-if-clause>,
    <s(2):is-c-ct-balanced-statement>,
    <s(3):is-PERC>,
    <s(4):is-c-ELSE>,
    <s(5):is-c-ct-balanced-statement>) v
   is-c-ct-unconditional-statement

(8) is-c-ct-unconditional-statement =
   (<s(1):is-PERC>,
    <s(2):is-0 v is-c-ct-labelist>,
    <s(3):is-c-ct-proper-statement>)

(9) is-c-ct-proper-statement =
   is-c-ct-do-group v is-c-ct-assignment-statement v
   is-c-ct-goto-statement v is-c-ct-null-statement v
   is-c-ct-activate-statement v is-c-ct-deactivate-statement v
   is-c-ct-include-statement
(10) \textit{is-c-ct-do-group} = \\
\textit{is-c-ct-simple-group} \lor \textit{is-c-ct-iterated-group}

(11) \textit{is-c-ct-simple-group} = \\
(<s(1): is-c-DO>, \\
<s(2): is-SEMIC>, \\
<s(3): is-c-ct-program>, \\
<s(4): is-c-ct-end-clause>)

(12) \textit{is-c-ct-end-clause} = \\
(<s(1): is-PERC>, \\
<s(2): is-Q \lor is-c-ct-labellist>, \\
<s(3): is-c-ENd>, \\
<s(4): is-Q \lor is-c-ct-identifier>, \\
<s(5): is-SEMIC>)

(13) \textit{is-c-ct-iterated-group} = \\
(<s(1): is-c-DO>, \\
<s(2): is-c-ct-do-specification>, \\
<s(3): is-SEMIC>, \\
<s(4): is-c-ct-program>, \\
<s(5): is-c-ct-end-clause>)
(14) \(\text{is-c-ct-no-specification} =\)
\[
(<s(1):\text{is-c-ct-simple-reference}>,
<s(2):\text{is-EQ}>,
<s(3):\text{is-c-ct-expression}>,
<s(4):\text{is-SEM}>
\]
\[
(<s(1):\text{is-c-BY}>,
<s(2):\text{is-c-ct-expression}>,
<s(3):\text{is-SEM}>
\]
\[
(<s(1):\text{is-c-TO}>,
<s(2):\text{is-c-ct-expression}>)\]
\[
(<s(1):\text{is-c-BY}>,
<s(2):\text{is-c-ct-expression}>)\]

(15) \(\text{is-c-ct-simple-reference} =\) \(\text{is-c-ct-identifier} \)

(16) \(\text{is-c-ct-assignment-statement} =\)
\[
(<s(1):\text{is-c-ct-simple-reference}>,
<s(2):\text{is-EQ}>,
<s(3):\text{is-c-ct-expression}>,
<s(4):\text{is-SEM}>)
\]

(17) \(\text{is-c-ct-expression} =\)
\[
\text{is-c-ct-expression-six} \lor (<s(1):\text{is-c-ct-expression}>,
<s(2):\text{is-OR}>,
<s(3):\text{is-c-ct-expression-six}>)
\]

(18) \(\text{is-c-ct-expression-six} =\)
\[
\text{is-c-ct-expression-five} \lor (<s(1):\text{is-c-ct-expression-six}>,
<s(2):\text{is-AND}>,
<s(3):\text{is-c-ct-expression-five}>)
\]
(19) \textit{is-c-ct-expression-five} = \\
\textit{is-c-ct-expression-four} v (\langle s(1) : \textit{is-c-ct-expression-five} \rangle, \\
\langle s(2) : \textit{is-c-ct-comparison-operator} \rangle, \\
\langle s(3) : \textit{is-c-ct-expression-four} \rangle) \\

(20) \textit{is-c-ct-comparison-operator} = \\
\textit{is-GT} v (\langle \text{elem}(1) : \textit{is-GT} \rangle, \\
\langle \text{elem}(2) : \textit{is-EQ} \rangle) v \\
\textit{is-EQ} v (\langle \text{elem}(1) : \textit{is-LT} \rangle, \\
\langle \text{elem}(2) : \textit{is-EQ} \rangle) v \\
\textit{is-LT} v (\langle \text{elem}(1) : \textit{is-NOT} \rangle, \\
\langle \text{elem}(2) : \textit{is-GT} \rangle) v \\

(\langle \text{elem}(1) : \textit{is-NOT} \rangle, \\
\langle \text{elem}(2) : \textit{is-EQ} \rangle) v \\

(\langle \text{elem}(1) : \textit{is-NOT} \rangle, \\
\langle \text{elem}(2) : \textit{is-LT} \rangle) \\

(21) \textit{is-c-ct-expression-four} = \\
\textit{is-c-ct-expression-three} v (\langle s(1) : \textit{is-c-ct-expression-four} \rangle, \\
\langle s(2) : (\langle \text{elem}(1) : \textit{is-OR} \rangle, \\
\langle \text{elem}(2) : \textit{is-OR} \rangle) \rangle, \\
\langle s(3) : \textit{is-c-ct-expression-three} \rangle) \\

(22) \textit{is-c-ct-expression-three} = \\
\textit{is-c-ct-expression-two} v (\langle s(1) : \textit{is-c-ct-expression-three} \rangle, \\
\langle s(2) : \textit{is-PLUS} v \textit{is-MINUS} \rangle, \\
\langle s(3) : \textit{is-c-ct-expression-two} \rangle) \\

(23) \textit{is-c-ct-expression-two} = \\
\textit{is-c-ct-expression-one} v (\langle s(1) : \textit{is-c-ct-expression-two} \rangle, \\
\langle s(2) : \textit{is-ASTER} v \textit{is-SLASH} \rangle, \\
\langle s(3) : \textit{is-c-ct-expression-one} \rangle) \\

(24) \textit{is-c-ct-expression-one} = \\
\textit{is-c-ct-primitive-expression} v (\langle s(1) : \textit{is-PLUS} v \textit{is-MINUS} v \textit{is-NOT} \rangle, \\
\langle s(2) : \textit{is-c-ct-expression-one} \rangle)
(25) \textit{is-c-ct-primitive-expression} =
\begin{align*}
&\quad (<s(1):is-LEFT-PAR>, \\
&\quad <s(2):is-c-ct-expression>, \\
&\quad <s(3):is-RIGHT-PAR>) \text{ v is-c-ct-reference v is-c-ct-constant}
\end{align*}

(26) \textit{is-c-ct-reference} =
\begin{align*}
&\quad (<s(1):is-c-ct-identifier>, \\
&\quad <s(2):is-0 v \\
&\quad (<s(1):is-LEFT-PAR>, \\
&\quad <s(2):is-c-ct-identifier>, \\
&\quad <s(3):is-RIGHT-PAR>) >)
\end{align*}

(27) \textit{is-c-ct-goto-statement} =
\begin{align*}
&\quad (<s(1):is-c-GOTO v \\
&\quad (<s(1):is-c-30>, \\
&\quad <s(2):is-c-TO>) >, \\
&\quad <s(2):is-c-ct-identifier>, \\
&\quad <s(3):is-SEMIC})
\end{align*}

(28) \textit{is-c-ct-null-statement} =
\begin{align*}
is-SEMIC
\end{align*}

(29) \textit{is-c-ct-return-statement} =
\begin{align*}
&\quad (<s(1):is-c-RETURN>, \\
&\quad <s(2):is-LEFT-PAR>, \\
&\quad <s(3):is-c-ct-expression>, \\
&\quad <s(4):is-RIGHT-PAR>, \\
&\quad <s(5):is-SEMIC})
\end{align*}

(30) \textit{is-c-ct-activate-statement} =
\begin{align*}
&\quad (<s(1):is-c-ACTIVATE>, \\
&\quad <s(2):is-c-ct-identifier>, \\
&\quad <s(3):is-SEMIC>)
\end{align*}
(31) is-c-ct-deactivate-statement =
(\langle s(1) : is-c-DEACTIVATE >,
\langle s(2) : (\langle s- del : is-COMMA >,
\langle s(1) : is-c-ct-identifier >,... ) >,
\langle s(3) : is-SEMIC >)

(32) is-c-ct-inclu de-statement =
(\langle s(1) : is-c-INCLUDE >,
\langle s(2) : (\langle s-del : is-COMMA >,
\langle s(1) : is-c-ct-library-specification >,... ) >,
\langle s(3) : is-SEMIC >)

(33) is-c-ct-library-specification =
(\langle s(1) : is-Q v is-c-ct-identifier >,
\langle s(2) : is-LEFT-PAR >,
\langle s(3) : is-c-ct-identifier >,
\langle s(4) : is-RIGHT-PAR > v is-c-ct-identifier

(34) is-c-ct-procedure =
(\langle s(1) : is-PERC >,
\langle s(2) : is-c-ct-entry-namelist >,
\langle s(3) : is-c-PROCEDURE >,
\langle s(4) : is-Q v is-c-ct-parameterlist >,
\langle s(5) : is-c-CHARACTER v is-c-FIXED >,
\langle s(6) : is-SEMIC >,
\langle s(7) : is-Q v is-c-ct-p-sentencelist >,
\langle s(8) : is-c-ct-end-clause >)

(35) is-c-ct-entry-namelist =
(\langle s(1) : (\langle s(1) : is-c-ct-identifier >,
\langle s(2) : is-COLON >) >,... )

(36) is-c-ct-parameterlist =
(\langle s(1) : is-LEFT-PAR >,
\langle s(2) : (\langle s-del : is-COMMA >,
\langle s(1) : is-c-ct-identifier >,... ) >,
\langle s(3) : is-RIGHT-PAR >)

(37) is-c-ct-p-sentencelist =
(\langle s(1) : is-c-ct-p-sentence >,... )
(38) is-c-ct-p-sentence =
    is-c-ct-p-statement v is-c-ct-p-declare-statement

(39) is-c-ct-p-statement =
    is-c-ct-p-if-statement v is-c-ct-p-unconditional-statement

(40) is-c-ct-p-if-statement =
    (\<s(1):is-c-ct-p-if-clause>,
     \<s(2):is-c-ct-p-statement>) v
    (\<s(1):is-c-ct-p-if-clause>,
     \<s(2):is-c-ct-p-balanced-statement>,
     \<s(3):is-c-ELSE>,
     \<s(4):is-c-ct-p-statement>)

(41) is-c-ct-p-if-clause =
    (\<s(1):is-Q v is-c-ct-labelist>,
     \<s(2):is-c-IF>,
     \<s(3):is-c-ct-expression>,
     \<s(4):is-c-THEN>)

(42) is-c-ct-p-balanced-statement =
    (\<s(1):is-c-ct-p-if-clause>,
     \<s(2):is-c-ct-p-balanced-statement>,
     \<s(3):is-c-ELSE>,
     \<s(4):is-c-ct-p-balanced-statement>) v
    is-c-ct-p-unconditional-statement

(43) is-c-ct-p-unconditional-statement =
    (\<s(1):is-Q v is-c-ct-labelist>,
     \<s(2):is-c-ct-proper-statement>)

(44) is-c-ct-p-proper-statement =
    is-c-ct-p-do-group v is-c-ct-assignment-statement v
    is-c-ct-qoto-statement v is-c-ct-null-statement v
    is-c-ct-return-statement

(45) is-c-ct-p-do-group =
    is-c-ct-p-simple-group v is-c-ct-p-iterated-group
(46) is-c-ct-p-simple-group =
   (<s(1):is-c-DO>,
    <s(2):is-SEMIC>,
    <s(3):is-Q v is-c-ct-p-sentencelist>,
    <s(4):is-c-ct-p-end-clause>)

(47) is-c-ct-p-end-clause =
   (<s(1):is-Q v is-c-ct-labelist>,
    <s(2):is-c-END>,
    <s(3):is-Q v is-c-ct-identifier>,
    <s(4):is-SEMIC>)

(48) is-c-ct-p-iterated-group =
   (<s(1):is-c-DO>,
    <s(2):is-c-ct-do-specification>,
    <s(3):is-SEMIC>,
    <s(4):is-Q v is-c-ct-p-sentencelist>,
    <s(5):is-c-ct-p-end-clause>)

(49) is-c-ct-p-declare-statement =
   (<s(1):is-Q v (<s(1):(<s(1):is-c-ct-identifier>,
               <s(2):is-COLON>),...),
    <s(2):is-c-DECLARE>,
    <s(3):is-c-ct-p-declarationlist>,
    <s(4):is-SEMIC>)

(50) is-c-ct-p-declarationlist =
   (<s(4):is-COMMA>,
    <s(1):is-c-ct-p-declaration>,...)

(51) is-c-ct-p-declaration =
    (<s(1):is-c-ct-identifier v
     (<s(1):is-LEFT-PAR>,
      <s(2):is-c-ct-p-declarationlist>,
      <s(3):is-RIGHT-PAR>),
     <s(2):is-Q v (<s(1):is-c-ct-p-attribute>,...))

(52) is-c-ct-p-attribute =
    is-c-FIXED v is-c-CHARACTER
(53) is-c-ct-declare-statement =
    (<s(1):is-PERC>,
     <s(2):is-QL v (<s(1):(<s(1):is-c-ct-identifier>,
                      <s(2):is-COLON>)>,...)>,
     <s(3):is-c-DECLARE>,
     <s(4):is-c-ct-declarationlist>,
     <s(5):is-SEMIC>)

(54) is-c-ct-declarationlist =
    (<s-de1:is-COMMA>,
     <s(1):is-c-ct-declaration>,...)}

(55) is-c-ct-declaration =
    (<s(1):is-c-ct-identifier v
     
     (<s(1):is-LEFT-PAR>,
      <s(2):is-c-ct-declarationlist>,
      <s(3):is-RIGHT-PAR>>),
     <s(2):is-QL v (<s(1):is-c-ct-attribute>,...)>)

(56) is-c-ct-attribute =
    is-c-FIXED v is-c-CHARACTER v is-c-ct-entry-attribute v
    is-c-c-returns-attribute

(57) is-c-ct-entry-attribute =
    (<s(1):is-c-ENTRY>,
     <s(2):is-QL v (<s(1):is-LEFT-PAR>,
                     <s(2):is-c-ct-descriptorlist>,
                     <s(3):is-RIGHT-PAR>>)>

(58) is-c-ct-descriptorlist =
    (<s(1):is-c-ct-descriptor>,
     <s(2):(<s(1):is-COMMA>,
            <s(2):is-c-ct-descriptorlist>)>)

(59) is-c-ct-descriptor =
    is-QL v is-c-FIXED v is-c-CHARACTER
(60) is-c-ct-returns-attribute =
    \langle s(1) : is-c-ct-RETURNS >,
    \langle s(2) : is-LEFT-P AR >,
    \langle s(3) : is-c-FIXED v is-c-CHARACTER >,
    \langle s(4) : is-RIGHT-P AR >)

(61) is-c-ct-identifier =
    \langle e lem(1) : is-c-ct-letter >,
    \langle e lem(2) : is-Q v ( \langle e lem(1) : is-c-ct-alphameric-character> , ... ) >

(62) is-c-ct-letter =
    is-A-CHAR v is-B-CHAR v is-C-CHAR v is-D-CHAR v is-E-CHAR v
    is-F-CHAR v is-G-CHAR v is-H-CHAR v is-I-CHAR v is-J-CHAR v
    is-K-CHAR v is-L-CHAR v is-M-CHAR v is-N-CHAR v is-O-CHAR v
    is-P-CHAR v is-Q-CHAR v is-R-CHAR v is-S-CHAR v is-T-CHAR v
    is-U-CHAR v is-V-CHAR v is-W-CHAR v is-X-CHAR v is-Y-CHAR v
    is-Z-CHAR v is-DOLLAR v is-COM-A T v is-NUMBER-SIGN

(63) is-c-ct-alphameric-character =
    is-c-ct-letter v is-c-ct-digit v is-BREAK

(64) is-c-ct-digit =
    is-0-CHAR v is-1-CHAR v is-2-CHAR v is-3-CHAR v is-4-CHAR v
    is-5-CHAR v is-6-CHAR v is-7-CHAR v is-8-CHAR v is-9-CHAR v

(65) is-c-ct-constant =
    is-c-ct-integer v is-c-ct-character-string v is-c-ct-bit-string

(66) is-c-ct-integer =
    \langle e lem(1) : is-0 v is-c-ct-integer >,
    \langle e lem(2) : is-c-ct-digit >

(67) is-c-ct-bit-string =
    \langle e lem(1) : is-APOSTR >,
    \langle e lem(2) : is-0 v ( \langle e lem(1) : is-c-ct-bit > , ... ) >,
    \langle e lem(3) : is-APOSTR >,
    \langle e lem(4) : is-3-CHAR >

(68) is-c-ct-bit =
    is-0-CHAR v is-1-CHAR
(69) is-c-ct-character-string =
    (elem(1):is-APOSTR>,
    (elem(2):is-Q v (elem(1):is-c-ct-string-character>,...))>
    (elem(3):is-APOSTR>)

(70) is-c-ct-string-character =
    is-c-ct-alphameic-character v is-BLANK v (elem(1):is-APOSTR>,
    elem(2):IS-APOSTR>) v
    is-EQ v is-PLUS v is-MINUS v is-ASTER v is-SLASH v is-LEFT-PAR v
    is-RIGHT-PAR v is-COMMA v is-POINT v is-SEMIC v is-COLON v is-AND v
    is-OR v is-NOT v is-GT v is-LT v is-QUEST v is-PERC v
    is-c-extralingual-character

(71) is-c-ct-text =
    is-Q v (elem(1):is-ASTER>,...)) v (elem(1):is-SLASH>,...)) v
    (elem(1):elem(1):(elem(1):is-Q v (elem(1):is-ASTER>,...)) v
    (elem(1):is-SLASH>,...))>
    (elem(2):is-c-ct-text-character v
    is-c-ct-string-part v
    is-c-ct-comment>)>
    (elem(2):is-Q v (elem(1):is-ASTER>,...))>
    (elem(1):is-SLASH>,...))>

(72) is-c-ct-text-character =
    is-c-ct-alphameic-character v is-BLANK v is-EQ v is-PLUS v is-MINUS v
    is-LEFT-PAR v is-RIGHT-PAR v is-COMMA v is-POINT v is-SEMIC v is-COLON v
    is-AND v is-OR v is-NOT v is-GT v is-LT v is-QUEST

(73) is-c-ct-string-part =
    (elem(1):is-APOSTR>,
    elem(2):is-Q v (elem(1):is-c-ct-string-part-char>,...))>
    elem(3):is-APOSTR>)
(74) is-c-ct-string-part-char =
   is-c-ct-alphanumeric-character v is-BLANK v is-EQ v is-PLUS v is-MINUS v
   is-ASTER v is-SLASH v is-LEFT-PAR v is-RIGHT-PAR v is-COMMA v is-POINT v
   is-SEMIC v is-COLON v is-AND v is-OR v is-NOT v is-GT v
   is-LT v is-QUEST v is-PERC v is-c-extralingual-character

(75) is-c-ct-comment =
   (<elem(1):is-SLASH>,
    <elem(2):is-ASTER>,
    <elem(3):is-Q v (<elem(1):(<elem(1):is-Q v (<elem(1):is-ASTER>,...>),
     <elem(2):is-c-ct-comment-symbol>) v
    is-SLASH>,...>),
    <elem(4):(<elem(1):is-ASTER>,...>),
    <elem(5):is-SLASH>)

(76) is-c-ct-comment-symbol =
   is-c-ct-alphanumeric-character v is-BLANK v is-EQ v is-PLUS v is-MINUS v
   is-LEFT-PAR v is-RIGHT-PAR v is-COMMA v is-POINT v is-SEMIC v is-COLON v
   is-APOSTR v is-AND v is-OR v is-NOT v is-GT v is-LT v
   is-QUEST v is-PERC v is-c-extralingual-character
5. THE TRANSLATOR

This chapter defines the translation from the abstract representation of a concrete ct-program into an abstract ct-program. This translation is performed by the function

\[ \text{ct-translate}(t) \]

which maps an object \( t \) satisfying the predicate is-c-ct-program (cf. 4-8(1)) onto an object satisfying the predicate is-et-program (cf. 6-1(1)).

In general, the translation of a part of an abstract represented concrete ct-program depends not only on this part itself but also on the context in which it occurs within the complete ct-program. Hence, the complete abstract represented ct-program \( t \) to be translated must be an argument of most of the functions of the translator. This argument is hidden, similar to the machine state \( \mathfrak{M} \) of the interpreter. Throughout this chapter the letter \( t \) denotes this hidden argument.

Instead of objects to be translated, which are components of \( t \), generally the selectors selecting them from \( t \) are specified as arguments of the functions. These selectors, called "pointers", are composed of selectors of the form \( \text{elem}(i) \) and \( s(i) \), \( i \) being integer values). They are usually named by the letters \( p, q, r \).

Both arguments together, the hidden argument \( t \) and an explicitly specified pointer \( p \), constitute all necessary information: A part of \( t \), namely \( p(t) \) and the context of this part within \( t \).

Note: All those functions, which handle components of \( t \) whose translation is context independent, have as argument these components themselves and not their pointers.

Metavariables

- \( t \): is-c-ct-program
- \( p, q, r \): pointers, i.e. selectors composed of \( \text{elem}(i), s(i) \), for is-intg-val(i), to be applied to \( t \)
- \( b \): is-c-ct-program\( \bowtie \)(t): pointer to a ct-program or to a ct-procedure
(1) \( \text{ct-translate}(t) = \)
\[
\begin{align*}
\text{is-c-ct-program}(t) & \quad \mu_0(<s-ct-decl-part:\text{mk-decl-part}(I)>; \\
<s-ct-text-part-list:\text{mk-text-part-list}(I)>)
\end{align*}
\]
\( T \rightarrow \text{error} \)

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

5.1 Construction of the Declaration-Part

(2) \( \text{mk-decl-part}(b) = \)
\[
\begin{align*}
\mu_0(\{<\text{id}:\text{mk-decl}([q \mid \text{is-attr-of}(r,q)]) > \mid r \in \text{decl-set}(b) & \\
\text{id} = \text{mk-ct-id}*t(t)\})
\end{align*}
\]
Ref.: \( \text{mk-ct-id} \ 2-3(8) \)

(3) \( \text{decl-set}(b) = \)
\[
\begin{align*}
\neg(\exists q,r)(\text{is-local-to}(b,q) & \& \text{is-local-to}(b,r) & \& \text{is-mult-decl}(q,r) \quad \rightarrow \quad \\
\{q \mid \text{is-local-to}(b,q) & \& \text{is-decl}(q) & \\
\text{(is-entry-cont}(q) \Rightarrow \neg(\exists r)(\text{is-entry-decl-cont}(r) & \& r(t) = q(t)))\})
\end{align*}
\]
\( T \rightarrow \text{error} \)

Note: This function yields a set consisting of all pointers pointing to identifiers being local to \( \text{b}(t) \) and which occur within declarations, or labellists, or entry-namelist provided that these entry names are not declared. The necessary multiple declarations check is also performed here.

(4) \( \text{is-decl}(p) = \)
\[
\text{is-entry-decl-cont}(p) \lor \text{is-var-decl-cont}(p) \lor \text{is-label-cont}(p) \lor \\
\text{is-entry-cont}(p)
\]

(5) \( \text{is-entry-decl-cont}(p) = \)
\[
\text{is-c-ct-identifier}^{\circ}p(t) & \& (\exists q,r,i)(\text{is-c-ct-declaration}^{\circ}q(t) & \\
\text{is-c-ct-entry-attribute}^{\circ}r(t) & \& q \Rightarrow p & \& r = s_1^*s_2^*q)
\]

(6) \( \text{is-var-decl-cont}(p) = \)
\[
\text{is-c-ct-identifier}^{\circ}p(t) & \& (\exists q,r,i)(\text{is-c-ct-declaration}^{\circ}q(t) & \\
\text{(is-c-FIXED}^{\circ}x(t) \lor \text{is-c-CHARACTER}^{\circ}r(t)) & q \Rightarrow p & \& r = s_1^*s_2^*q)
\]

5.1
(7) \(\text{is-label-cont}(p) =\)

\[
is-c-ct-identifier\cdot p(t) \land \\
(\exists q, i)(\text{is-c-ct-unconditional-statement}\cdot q(t) \lor \text{is-c-ct-if-clause}\cdot q(t) \lor \\
\text{is-c-ct-end-clause}\cdot q(t)) \land p = s_1^*s_1^*s_2^*q \lor \\
(\text{is-c-ct-p-unconditional-statement}\cdot q(t) \lor \text{is-c-ct-p-if-clause}\cdot q(t) \lor \\
\text{is-c-ct-p-end-clause}\cdot q(t)) \land p = s_1^*s_1^*s_1^*q)
\]

(8) \(\text{is-entry-cont}(p) =\)

\[
is-c-ct-identifier\cdot p(t) \land \\
(\exists q, i)(\text{is-c-ct-procedure}\cdot q(t) \land p = s_1^*s_1^*s_2^*q)
\]

(9) \(\text{is-local-to}(b, q) =\)

\[
is-c-ct-program\cdot b(t) \rightarrow b \Rightarrow q \land \neg s_2^*b \Rightarrow q \\
is-c-ct-program\cdot b(t) \rightarrow b \Rightarrow q \land \\
(\forall r)(\text{is-c-ct-procedure}\cdot r(t) \land b \Rightarrow r \Rightarrow q \Rightarrow s_2^*r \Rightarrow q)
\]

(10) \(\text{is-mult-decl}(p, q) =\)

\[
p \neq q \land \text{is-decl}(p) \land \text{is-decl}(q) \land p(t) = q(t) \land \\
\neg(\text{is-entry-decl-cont}(p) \land \text{is-entry-cont}(q))
\]

(11) \(\text{is-attr-of}(p, at) =\)

\[
\text{is-entry-decl-cont}(p) \rightarrow \\
is-c-ct-attribute\cdot at(t) \land (\exists q, i)(\text{is-c-ct-declaration}\cdot q(t) \land \\
q \Rightarrow p \land at = s_1^*s_2^*q) \lor \text{is-entry-cont}(at) \land at(t) = p(t)
\]

\[
is-var-decl-cont(p) \rightarrow \\
is-c-ct-attribute\cdot at(t) \land (\exists q, i)(\text{is-c-ct-declaration}\cdot q(t) \land \\
q \Rightarrow p \land at = s_1^*s_2^*q)
\]

\[
is-label-cont(p) \rightarrow \text{is-LAB}(at)
\]

\[
is-entry-cont(p) \rightarrow at = p
\]

for: \(\text{is-decl}(p), \text{is-LAB}(at), \text{is-entry-cont}(at) \lor \text{is-c-ct-attribute}\cdot at(t)\)
(12) \( \text{mk-decl}(\text{ats}) = \)
\[
(\exists p) (p \in \text{ats} \land \text{is-LAB}(p)) \quad \rightarrow \quad \text{LAB}
\]
\[
(\exists p) (p \in \text{ats} \land \text{is-c-FIXED}(p)) \quad \rightarrow \quad \text{INTG}
\]
\[
(\exists p) (p \in \text{ats} \land \text{is-c-CHARACTER}(p)) \quad \rightarrow \quad \text{CHAR}
\]
\[
(\exists p) (p \in \text{ats} \land \text{is-c-ct-entry-attribute}(p)) \quad \land
\]
\[
(\exists q) (q \in \text{ats} \land (\text{is-c-FIXED}(q) \lor \text{is-c-CHARACTER}(q)))
\]
\[
\mu_0 (<s-ct-par-list \cdot s-ct-entry-decl:\n(\forall p) (p \in \text{ats} \land \text{is-c-ct-entry-attribute}(p) \lor \text{is-c-FIXED}(p)) \quad \rightarrow \quad <>),
\quad T \quad \rightarrow \quad \text{mk-descr-list}(<s-ct-par-list \cdot s-ct-entry-decl: \text{mk-ret-type}((\forall q) (q \in \text{ats} \land \text{is-c-ct-returns-attribute}(q))))>,
\quad <s-ct-body:((\exists q) (q \in \text{ats} \land \text{is-c-ct-returns-attribute}(q)) \rightarrow 
\quad \text{proc-trans}((\forall q) (q \in \text{ats} \land \text{is-c-ct-returns-attribute}(q))),
\quad T \quad \rightarrow \quad \emptyset >)
\]
\[
(\exists p,q) (p \neq q \lor p,q \in \text{ats})
\]
\[
\mu_0 (<s-ct-body:\text{proc-trans}((\forall q) (q \in \text{ats} \land \text{is-c-ct-returns-attribute}(q))>)
\]
\[
T \quad \rightarrow \quad \text{error}
\]

(13) \( \text{mk-descr-list}(\text{p}) = \text{mk-descr-list-l} (s_2 \cdot s_2 \cdot p(t),\leftrightarrow) \)

for: \( \text{is-c-ct-entry-attribute}(p(t)) \land \neg \text{is-c-FIXED}(p(t)) \)

(14) \( \text{mk-descr-list-l}(\text{cdl},\text{adl}) = \)
\[
\text{is-c-FIXED}(\text{cdl}) \quad \rightarrow \quad \text{adl} \land <\text{mk-descr-s_1}(\text{cdl})>
\]
\[
T \quad \rightarrow \quad \text{mk-descr-list-l}(s_2 \cdot s_2 \cdot cdl,\text{adl} \land <\text{mk-descr-s_1}(\text{cdl})>)
\]

for: \( \text{is-c-ct-descriptor-list}(\text{cdl}), \)
\[\text{is-c-ct-descriptor-list}(\text{adl}) \]

(15) \( \text{mk-descr}(\text{x}) = \)
\[
\text{is-c-FIXED}(\text{x}) \quad \rightarrow \quad *\quad \text{INTG}
\]
\[
\text{is-c-CHARACTER}(\text{x}) \quad \rightarrow \quad \text{CHAR}
\]

(16) \( \text{mk-ret-type}(\text{p}) = \text{mk-descr-s_3}(\text{p(t)}) \)

for: \( \text{is-c-ct-returns-attribute}(p(t)) \)
(17) proc-trans(p) =
\[
\begin{align*}
p = \text{en}_1 & \rightarrow \text{mk-body}((\lambda q)(\text{is-c-ct-procedure}\cdot q(t) \& q \Rightarrow p)) \\
T & \rightarrow \text{mk-ct-id}\cdot \text{en}_1(t)
\end{align*}
\]
where: \(\text{en}_1 = s_1 \cdot s_1((\lambda q)(\text{is-c-ct-entry-namelist}\cdot q(t) \& q \Rightarrow p))\)

for: \(\text{is-entry-cont}(p)\)

Ref.: mk-ct-id 2-3(8)

Note: Only the first identifier of the entry-namelist is connected with the body in the declaration-part. All other entry-names will be associated with the first identifier of the entry-namelist.

(18) mk-body(p) =
\[
(\exists l)(s_1 \cdot s_1 \cdot s_2 \cdot p(t) = s_4 \cdot s_8 \cdot p(t)) \rightarrow \\
\mu_0(<\text{s-ct-par-list: LIST mk-ct-id}\cdot s_1 \cdot s_2 \cdot s_4 \cdot p(t)>, \\
<s\text{-ct-ret-type: mk-descr}\cdot s_5 \cdot p(t)>, \\
<s\text{-ct-decl-part: mk-decl-part}(p)>, \\
<s\text{-ct-text-part-list: trans-p-selist}(s_7 \cdot p)>) \\
\text{trans-p-end-clause}(s_2 \cdot s_8 \cdot p)>) \\
T \rightarrow \text{error}
\]

for: \(\text{is-c-ct-procedure}\cdot p(t)\)

Ref.: mk-ct-id 2-3(8)

(19) trans-p-selist(p) = trans-p-selist-l(p,l)

for: \(\text{is-c-ct-p-sentencelist}\cdot p(t) \lor \text{is-}\varnothing\cdot p(t)\)

(20) trans-p-selist-l(p,i) =
\[
\text{is-}\varnothing \cdot s_1 \cdot p(t) \rightarrow <> \\
is\text{-c-ct-p-declare-statement}\cdot s_1 \cdot p(t) \rightarrow \text{trans-p-selist-l}(p,i+1) \\
T \rightarrow \mu_0(s\text{-ct-st\cdot elem(1): trans-p-st}(s_1 \cdot p)) \text{trans-p-selist-l}(p,i+1)
\]

for: \(\text{is-c-ct-p-sentencelist}\cdot p(t) \lor \text{is-}\varnothing\cdot p(t)\)

\(\text{is-intg-val}(i)\)
(21) \(\text{trans-p-st}(p) =\)
\[
is\text{-c-ct-p-if-clause} \cdot s_1 \cdot p(t) \rightarrow \text{trans-p-if-st}(p) \]
\[
(\text{is-c-ct-p-simple-group} \cdot s_2 \cdot p(t) \supset (\exists i) (s_1 \cdot s_1 \cdot s_1 \cdot p(t) = s_3 \cdot s_4 \cdot s_2 \cdot p(t))) \land \]
\[
(\text{is-c-ct-p-iterated-group} \cdot s_2 \cdot p(t) \supset (\exists i) (s_1 \cdot s_1 \cdot s_1 \cdot p(t) = s_3 \cdot s_5 \cdot s_2 \cdot p(t))) \rightarrow \]
\[
\mu_{o}(s\text{-ct-label-list}\cdot \text{trans-label-list}\cdot s_1 \cdot p(t)), \]
\[
\mu_{o}(s\text{-ct-prop-st} \cdot (\text{is-c-ct-p-do-group} \cdot s_2 \cdot p(t) \rightarrow \text{trans-p-group}(s_2 \cdot p),
T \rightarrow \text{trans-prop-st}(s_2 \cdot p)))
\]
\[T \rightarrow \text{error}\]

for: \(\text{is-c-ct-p-statement}\cdot p(t)\)

Ref.: trans-label-list 5-10(39), trans-prop-st 5-10(40)

(22) \(\text{trans-p-if-st}(p) =\)
\[
\mu_{o}(s\text{-ct-label-list}\cdot \text{trans-label-list}\cdot s_1 \cdot s_1 \cdot p(t)), \]
\[
\mu_{o}(s\text{-ct-stmt} : \text{IF}), \]
\[
\mu_{o}(s\text{-ct-decision} : \text{trans-expr}(s_3 \cdot s_1 \cdot p)), \]
\[
\mu_{o}(s\text{-ct-st} \cdot s\text{-ct-then} : \text{trans-p-st}(s_2 \cdot p)), \]
\[
\mu_{o}(s\text{-ct-st} \cdot s\text{-ct-else} : \text{trans-else-p-st}(s_4 \cdot p)))
\]

for: \(\text{is-c-ct-p-if-statement}\cdot p(t) \lor \text{is-c-ct-p-balanced-statement}\cdot p(t)\)

Ref.: trans-label-list 5-10(39), trans-expr 5-12(43)

(23) \(\text{trans-else-p-st}(p) =\)
\[
is\cdot \emptyset \cdot p(t) \rightarrow \mu_{o}(s\text{-ct-label-list} : \emptyset), \]
\[
is\text{-ct-stmt} \cdot s\text{-ct-prop-st} : \text{NULL}
\]
\[T \rightarrow \text{trans-p-st}(p)\]

for: \(\text{is-c-ct-p-statement}\cdot p(t) \lor \text{is-\emptyset}\cdot p(t)\)

Ref.: trans-do-spec 5-9(36)

(24) \(\text{trans-p-group}(p) =\)
\[
is\text{-c-ct-p-simple-group} \cdot p(t) \rightarrow \]
\[
\text{trans-p-selist}(s_3 \cdot p) \land \text{trans-p-end-clause}(s_1 \cdot s_4 \cdot p)
\]
\[T \rightarrow \mu_{o}(s\text{-ct-iteration} : \text{trans-do-spec}(s_2 \cdot p)), \]
\[
\mu_{o}(s\text{-ct-text-part-list} : \text{trans-p-selist}(s_4 \cdot p)) \land \]
\[
\text{trans-p-end-clause}(s_1 \cdot s_5 \cdot p))
\]

for: \(\text{is-c-ct-p-do-group}\cdot p(t)\)

Ref.: trans-do-spec 5-9(36)
trans-p-end-clause(p) =
\[ \mu_0(<s-ct-st: \mu_0(<s-ct-label-list: \text{trans-label-list}*p(t)>,
\text{trans-label-list}*p(t)>) >, <s-ct-stmt*s-ct-prop-st: \text{NULL} >) >) \]

for: is-c-ct-labellist*p(t) v is-\(\emptyset\)*p(t)

Ref.: trans-label-list 5-10(39)

5.2 Construction of the Text-Part-List

mk-text-part-list(p) = mk-tpl(p,< >,< >)

for: is-c-ct-program*p(t)

mk-tpl(p,tpl,text) =
\[ \text{is-\(\emptyset\)*p(t) & is-\(\emptyset\)>(text) \rightarrow tpl} \]
\[ \text{is-c-ct-text*p(t) \rightarrow conc-tp(null_{\text{st}}, \text{lin-3} *p(t) \text{^ text}, tpl)} \]
\[ \text{is-c-ct-procedure*s_2*p(t) \rightarrow} \]
\[ \text{mk-tpl(s_1*p,tpl, \text{BLANK} ^ lin-3 *s_3*p(t) ^ text), < >)} \]

T \rightarrow mk-tpl(s_1*p,conc-tp(trans-se(s_2*p),\text{lin-3} *s_3*p(t) ^ text, tpl), < >)

where: null_{\text{st}} = \mu_0(<s-ct-label-list:< >, <s-ct-stmt*s-ct-prop-st: \text{NULL} >)

for: is-c-ct-program*p(t)

is-c-ct-text-part-list(tpl)

is-char-val-list(text)

Ref.: lin-3 4-3(6), is-char-val 6-4(46)

Note: This function forms a text-part-list according to the predicate is-c-ct-text-part-list of the abstract syntax (cf. 6-2 (18)).

Ct-procedures are replaced by the character-value "BLANK" because they are already contained in the declaration-part of the abstract program. Since also the first text-part of the text-part-list must contain an abstract ct-statement, an abstract ct-null statement is added if necessary.
\[(32) \text{trans-if-st}(p) = \mu_0(\langle s-ct-label-list:trans-label-list*s_2*s_1*p(t)\rangle, \langle s-ct-prop-st:\mu_0(\langle s-ct-stmt:IF\rangle, \langle s-ct-decision:trans-expr(s_4*s_1*p)\rangle, \langle s-ct-st*s-ct-then:trans-st(s_2*p)\rangle, \langle s-ct-st*s-ct-else:trans-else-st(s_5*p)\rangle)\rangle) \]

for: is-c-ct-if-statement*p(t) v is-c-ct-balanced-statement*p(t)

\[(33) \text{trans-else-st}(p) = \]

\[\text{is-} \O \text{p(t) } \longrightarrow \mu_0(\langle s-ct-label-list:\langle\rangle, \langle s-ct-stmt*s-ct-prop-st:NULL\rangle) \]

\[T \longrightarrow \text{trans-st}(p) \]

for: is-c-ct-statement*p(t) v is-\O p(t)

\[(34) \text{trans-group}(p) = \]

\[\text{is-c-ct-simple-group*p(t) } \longrightarrow \]

\[\text{mk-text-part-list}(s_3*p) \land \langle\text{trans-end-clause}(s_2*s_4*p)\rangle \]

\[T \longrightarrow \mu_0(\langle s-ct-iteration:trans-do-spec\langle s_2*p\rangle, \langle s-ct-text-part-list:mk-text-part-list(s_4*p) \land \langle\text{trans-end-clause}(s_2*s_5*p)\rangle\rangle) \]

for: is-c-ct-do-group*p(t)

\[(35) \text{trans-end-clause}(p) = \mu_0(\langle s-ct-st:\mu_0(\langle s-ct-label-list:trans-label-list*p(t)\rangle, \langle s-ct-stmt*s-ct-prop-st:NULL\rangle)\rangle, \langle s-ct-text:\langle\rangle\rangle) \]

for: is-c-ct-labellist*p(t) v is-\O p(t)

\[(36) \text{trans-do-spec}(p) = \mu_0(\langle s-ct-var:trans-siref*s_1*p(t)\rangle, \langle s-ct-init-expr:trans-expr(s_3*p)\rangle, \langle s-ct-by-expr:mk-by-expr(s_4*p)\rangle, \langle s-ct-to-expr:mk-to-expr(s_4*p)\rangle) \]

for: is-c-ct-do-specification*p(t)
(37) \[\text{mk-by-expr}(p) = \]
\[
\begin{align*}
\text{is-c-BY}^{*}s_1^{*}p(t) & \rightarrow \text{trans-expr}(s_2^{*}p) \\
\text{is-c-BY}^{*}s_3^{*}p(t) & \rightarrow \text{trans-expr}(s_2^{*}s_3^{*}p) \\
\text{--is-}Q^{*}p(t) & \rightarrow \mu_o(\langle s\text{-ct-da:INTG},<s\text{-ct-vr:ct-represent(INTG,1)>}) \\
T & \rightarrow \emptyset
\end{align*}
\]

Ref.: ct-represent 2-2(5)

(38) \[\text{mk-to-expr}(p) = \]
\[
\begin{align*}
\text{is-c-TO}^{*}s_1^{*}p(t) & \rightarrow \text{trans-expr}(s_2^{*}p) \\
\text{is-c-TO}^{*}s_3^{*}p(t) & \rightarrow \text{trans-expr}(s_2^{*}s_3^{*}p) \\
T & \rightarrow \emptyset
\end{align*}
\]

Ref.: ct-represent 2-3(3)

(39) \[\text{trans-label-list}(x) = \]
\[
\begin{align*}
\text{stlength}(i) & \\
\text{LIST mk-ct-id}s_1^{*}s_1(x) \\
\text{for: is-c-ct-labellist}(x)
\end{align*}
\]

Ref.: mk-ct-id 2-3(3)

(40) \[\text{trans-prop-st}(p) = \]
\[
\begin{align*}
\text{is-c-ct-assignment-statement}^{*}p(t) & \rightarrow \\
\mu_o(\langle s\text{-ct-stmt:ASSIGN},<s\text{-ct-lp:trans-sizef}s_1^{*}p(t)>,<s\text{-ct-rt:trans-expr}(s_3^{*}p)>) \\
\text{is-c-ct-goto-statement}^{*}p(t) & \rightarrow \mu_o(\langle s\text{-ct-stmt:GOTO},<s\text{-ct-label:mk-ct-id}s_2^{*}p(t)>) \\
\text{is-c-ct-null-statement}^{*}p(t) & \rightarrow \mu_o(\langle s\text{-ct-stmt:NULL}>) \\
\text{is-c-ct-activate-statement}^{*}p(t) & \rightarrow \\
\mu_o(\langle s\text{-ct-stmt:ACT},<s\text{-ct-id-list:LIST mk-ct-id}s_1^{*}s_2^{*}p(t>)i_{=1}^{k}\rangle \\
\text{is-c-ct-deactivate-statement}^{*}p(t) & \rightarrow \\
\mu_o(\langle s\text{-ct-stmt:DEACT},<s\text{-ct-id-list:LIST mk-ct-id}s_1^{*}s_2^{*}p(t>)i_{=1}^{l}\rangle \\
\text{cont'd}
\end{align*}
\]
\textbf{is-c-ct-include-statement}\{p(t)\} \rightarrow \\
\mu_0(\langle s-ct-stmt:INCL\rangle, \\
\langle s-ct-id-list:LIST\ trans-lib-spec(s_1^*s_2^*p)\rangle) \\
\textbf{is-c-ct-return-statement}\{p(t)\} \rightarrow \\
\mu_0(\langle s-ct-stmt:RETURN\rangle, \\
\langle s-ct-expr:trans-expr(s_3^*p)\rangle) \\
\text{where: } \lg_p = \text{length} s_2^*p(t) \\
\text{Ref.: mk-ct-id 2-3(A)} \\
\textbf{(41) trans-siref}(x) = \\
\mu_0(\langle s-ct-id:mk-ct-id(x)\rangle, \\
\langle s-ct-arg-list:<>\rangle) \\
\text{for: is-c-ct-identifier}\{x\} \\
\text{Ref.: mk-ct-id 2-3(A)} \\
\textbf{(42) trans-lib-spec}(p) = \\
\neg is-\emptyset s_1^*p(t) \rightarrow \mu_0(\langle s-ct-id-1:mk-ct-id,s_1^*p(t)\rangle, \\
\langle s-ct-id-2:mk-ct-id,s_3^*p(t)\rangle) \\
\neg is-\emptyset s_3^*p(t) \rightarrow \mu_0(\langle s-ct-id-1:*,s_3^*p(t)\rangle, \\
\langle s-ct-id-2:mk-ct-id,s_3^*p(t)\rangle) \\
T \rightarrow \mu_0(\langle s-ct-id-1:mk-ct-id,p(t)\rangle, \\
\langle s-ct-id-2:*,s_3^*p(t)\rangle) \\
\text{for: is-c-ct-library-specification}\{p(t)\} \\
\text{Ref.: mk-ct-id 2-3(8)}
(43) \( \text{trans-expr}(p) = \)

\[
\begin{align*}
\text{is-c-ct-reference} & \cdot p(t) \\
\mu_0 & \left( <s-ct-id: mk-ct-id \cdot s_1 \cdot p(t)> \right) \\
\text{is-c-ct-constant} & \cdot p(t) \\
\text{is-LEFT-PAR} & \cdot s_1 \cdot p(t) \\
\mu_0 & \left( <s-ct-opd: \text{trans-expr}(s_2 \cdot p) > \right) \\
\text{is-c-ct-expression} & \cdot p(t) \\
\end{align*}
\]

where: \( \lg_p = \text{length} \cdot s_2 \cdot p(t) \)

for: \( \text{is-c-ct-expression} \cdot p(t) \)

Ref.: \( \text{mk-ct-id } 2-2(8) \)

(44) \( \text{trans-const}(x) = \)

\[
\begin{align*}
\text{is-c-ct-integer}(x) & -\rightarrow \\
\mu_0 & \left( <s-ct-da: \text{INTG}, \\
\text{is-c-ct-character-string}(x) & -\rightarrow \\
\mu_0 & \left( <s-ct-da: \text{CHAR}, \\
\text{is-c-ct-bit-string}(x) & -\rightarrow \\
\mu_0 & \left( <s-ct-da: \text{BIT}, \\
\end{align*}
\]

where: \( \text{char}_1x = (x_1 < \text{APOSTR}, \text{APOSTR} \rightarrow \text{APOSTR}, T \rightarrow x_1) \)

\( x_1 = \text{elem}_1 \cdot \text{elem}(2, x) \)

for: \( \text{is-c-ct-constant}(x) \)

Ref.: \( \text{ct-represent } 2-2(5) \)
(45) \( \text{intg-val}(x) = \)
\[
\begin{align*}
\text{is-0 (} x \text{)} & \rightarrow 0 \\
\text{T} & \rightarrow 10 \cdot \text{intg-val} \cdot \text{elem}(1,x) + \text{digit-val} \cdot \text{elem}(2,x)
\end{align*}
\]
for: \( \text{is-c-ct-integer}(x) \)

(46) \( \text{digit-val}(x) = \)
\[
\begin{align*}
\text{is-0-CHAR}(x) & \rightarrow 0 \\
\text{is-1-CHAR}(x) & \rightarrow 1 \\
\text{is-2-CHAR}(x) & \rightarrow 2 \\
\text{is-3-CHAR}(x) & \rightarrow 3 \\
\text{is-4-CHAR}(x) & \rightarrow 4 \\
\text{is-5-CHAR}(x) & \rightarrow 5 \\
\text{is-6-CHAR}(x) & \rightarrow 6 \\
\text{is-7-CHAR}(x) & \rightarrow 7 \\
\text{is-8-CHAR}(x) & \rightarrow 8 \\
\text{is-9-CHAR}(x) & \rightarrow 9
\end{align*}
\]

(47) \( \lg(x) = \)
\[
\begin{align*}
\text{is-list}(x) & \rightarrow \text{length}(x) \\
\text{T} & \rightarrow 0
\end{align*}
\]

(48) \( \text{trans-infix}(x) = \)
\[
\begin{align*}
\text{is-PLUS}(x) & \rightarrow \text{ADD} \\
\text{is-MINUS}(x) & \rightarrow \text{SUBTR} \\
\text{is-ASTER}(x) & \rightarrow \text{MULT} \\
\text{is-SLASH}(x) & \rightarrow \text{DIV} \\
x = \langle \text{OR}, \text{OR} \rangle & \rightarrow \text{CAT} \\
x = \langle \text{GT}, \text{EQ} \rangle \lor x = \langle \text{NOT}, \text{LT} \rangle & \rightarrow \text{GE} \\
x = \langle \text{LT}, \text{EQ} \rangle \lor x = \langle \text{NOT}, \text{GT} \rangle & \rightarrow \text{LE} \\
x = \langle \text{NOT}, \text{EQ} \rangle & \rightarrow \text{NE} \\
\text{T} & \rightarrow x
\end{align*}
\]
6. ABSTRACT SYNTAX

This chapter defines the predicate is-ct-program, giving the structure of abstract ct-programs as produced by the function ct-translate defined in chapter 5. However, the range of the function ct-translate is a subset of the set of all abstract ct-programs.

The notation used in the definition of the abstract syntax is taken from /6/, 12.1.

(1) is-ct-program = (is-ct-decl-part, is-ct-text-part-list)

(2) is-ct-decl-part = ({id: is-ct-decl} || is-id(id))

Note: The predicate is-id characterizes an infinite class of elementary objects, called "abstract identifiers". (cf. 2.2).

(3) is-ct-decl = is-ct-prop-var v is-ct-entry v is-LAB

(4) is-ct-prop-var = is-INTG v is-CHAR

(5) is-ct-entry = (is-ct-entry-dec1: is-ct-entry-decl) v
  (is-ct-body: is-ct-body v is-id) v
  (is-ct-entry-dec1: is-ct-entry-decl, is-ct-body: is-ct-body v is-id)

(6) is-ct-entry-decl = (is-ct-par-list: is-ct-descr-list, is-ct-ret-type: is-INTG v is-CHAR)

(7) is-ct-descr = is-INTG v is-CHAR v is- *

(8) is-ct-body = (is-ct-par-list: is-id-list, is-ct-decl-part: is-ct-p-decl-part, is-ct-ret-type: is-INTG v is-CHAR, is-ct-text-part-list: is-ct-p-text-part-list)

(9) is-ct-p-decl-part = ({id: is-ct-p-decl} || is-id(id))

(10) is-ct-p-decl = is-ct-prop-var v is-LAB
(11) is-ct-p-text-part = (s-ct-st:is-ct-p-st)

(12) is-ct-p-st = (s-ct-label-list:is-id-list, 
    s-ct-prop-st:is-ct-p-prop-st)

(13) is-ct-p-prop-st = is-ct-p-group \or is-ct-p-text-part-list \or is-ct-p-if-st \or 
    is-ct-assign-st \or is-ct-goto-st \or is-ct-null-st \or 
    is-ct-return-st

(14) is-ct-p-group = (s-ct-iteration:is-ct-iteration, 
    s-ct-text-part-list:is-ct-p-text-part-list)

(15) is-ct-iteration = (s-ct-var:is-ct-simple-var, 
    s-ct-init-expr:is-ct-expr, 
    s-ct-by-expr:is-ct-expr \or \emptyset, 
    s-ct-to-expr:is-ct-expr \or \emptyset)

(16) is-ct-simple-var = (s-ct-id:is-id, 
    s-ct-arg-list:is-\langle\rangle)

(17) is-ct-p-if-st = (s-ct-stmt:is-IF, 
    s-ct-decision:is-ct-expr, 
    s-ct-then:is-ct-p-text-part, 
    s-ct-else:is-ct-p-text-part)

(18) is-ct-text-part = (s-ct-st:is-ct-st, 
    s-ct-text:is-char-val-list)

(19) is-ct-st = (s-ct-label-list:is-id-list, 
    s-ct-prop-st:is-ct-prop-st)

(20) is-ct-prop-st = is-ct-group \or is-ct-text-part-list \or is-ct-if-st \or 
    is-ct-assign-st \or is-ct-declare-st \or is-ct-goto-st \or 
    is-ct-null-st \or is-ct-act-st \or is-ct-deact-st \or 
    is-ct-include-st

(21) is-ct-group = (s-ct-iteration:is-ct-iteration, 
    s-ct-text-part-list:is-ct-text-part-list)
(22) \( \text{is-ct-if-st} = (\langle \text{ct-stmt:is-IF} \rangle, \)
\( \langle \text{ct-decision:is-ct-expr} \rangle, \)
\( \langle \text{ct-then:is-ct-if-text-part} \rangle, \)
\( \langle \text{ct-else:is-ct-if-text-part} \rangle ) \)

(23) \( \text{is-ct-if-text-part} = (\langle \text{ct-st:is-ct-st} \rangle ) \)

(24) \( \text{is-ct-assign-st} = (\langle \text{ct-stmt:is-ASSIGN} \rangle, \)
\( \langle \text{ct-lp:is-ct-simple-var} \rangle, \)
\( \langle \text{ct-rp:is-ct-expr} \rangle ) \)

(25) \( \text{is-ct-expr} = \text{is-ct-infix-expr} \lor \text{is-ct-prefix-expr} \lor \text{is-ct-paren-expr} \lor \)
\( \text{is-ct-ref} \lor \text{is-ct-const} \)

(26) \( \text{is-ct-infix-expr} = (\langle \text{ct-op:is-ct-infix-op} \rangle, \)
\( \langle \text{ct-opd-1:is-ct-expr} \rangle, \)
\( \langle \text{ct-opd-2:is-ct-expr} \rangle ) \)

(27) \( \text{is-ct-infix-op} = \text{is-CAT} \lor \text{is-ct-bit-op} \lor \text{is-ct-comp-op} \lor \text{is-ct-arith-op} \)

(28) \( \text{is-ct-bit-op} = \text{is-OR} \lor \text{is-AND} \)

(29) \( \text{is-ct-comp-op} = \text{is-GT} \lor \text{is-GE} \lor \text{is-EQ} \lor \text{is-LE} \lor \text{is-LT} \lor \text{is-NE} \)

(30) \( \text{is-ct-arith-op} = \text{is-ADD} \lor \text{is-SUBTR} \lor \text{is-MULT} \lor \text{is-DIV} \)

(31) \( \text{is-ct-prefix-expr} = (\langle \text{ct-op:is-ct-prefix-op} \rangle, \)
\( \langle \text{ct-opd:is-ct-expr} \rangle ) \)

(32) \( \text{is-ct-prefix-op} = \text{is-PLUS} \lor \text{is-MINUS} \lor \text{is-NOT} \)

(33) \( \text{is-ct-paren-expr} = (\langle \text{ct-opd:is-ct-expr} \rangle ) \)

(34) \( \text{is-ct-ref} = (\langle \text{ct-id:is-id} \rangle, \)
\( \langle \text{ct-arg-list:is-ct-expr-list} \rangle ) \)

(35) \( \text{is-ct-const} = \text{is-ct-intg} \lor \text{is-ct-string} \)

(36) \( \text{is-ct-intg} = (\langle \text{ct-da:is-INTG} \rangle, \)
\( \langle \text{ct-vr:is-ct-vr} \rangle ) \)

Note: The predicate is-ct-vr characterizes a class of elementary objects
called "value representations". (cf. 2.2).
(37) is-ct-string = (<s-ct-da:is-CHAR v is-BIT>,
                 <s-ct-vr:is-ct-vr-list>)

(38) is-ct-declare-st = (<s-ct-stmt:is-DCL>,
                         <s-ct-id-list:is-id-list>)

(39) is-ct-goto-st = (<s-ct-stmt:is-GOTO>,
                    <s-ct-label:is-id>)

(40) is-ct-null-st = (<s-ct-stmt:is-NULL>)

(41) is-ct-return-st = (<s-ct-stmt:is-RETURN>,
                         <s-ct-expr:is-ct-expr>)

(42) is-ct-act-st = (<s-ct-stmt:is-ACT>,
                    <s-ct-id-list:is-id-list>)

(43) is-ct-deact-st = (<s-ct-stmt:is-DEACT>,
                      <s-ct-id-list:is-id-list>)

(44) is-ct-include-st = (<s-ct-stmt:is-INCL>,
                         <s-ct-id-list:is-ct-id-pair-list>)

(45) is-ct-id-pair = (<s-ct-id-1:is-id v is->>,
                    <s-ct-id-2:is-id v is->>)

(46) is-char-val = is-alpham-char v is-del-char

(47) is-alpham-char = is-letter v is-digit v is-BREAK

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

(49) is-digit = is-O-CHAR v is-1-CHAR v is-2-CHAR v is-3-CHAR v is-4-CHAR v
             is-5-CHAR v is-6-CHAR v is-7-CHAR v is-8-CHAR v is-9-CHAR
(5o) \( \text{is-del-char} = \text{is-BLANK} \lor \text{is-APOSTR} \lor \text{is-EQ} \lor \text{is-PLUS} \lor \text{is-MINUS} \lor \text{is-ASTER} \lor \text{is-SLASH} \lor \text{is-LEFT-PAR} \lor \text{is-RIGHT-PAR} \lor \text{is-COMMA} \lor \text{is-POINT} \lor \text{is-SEMIC} \lor \text{is-COLON} \lor \text{is-AND} \lor \text{is-OR} \lor \text{is-NOT} \lor \text{is-GT} \lor \text{is-LT} \lor \text{is-QUEST} \lor \text{is-PERC} \lor \text{is-extralingual-char} \)

Note: The predicate \( \text{is-extralingual-char} \) is implementation-defined. It is equivalent to the predicate \( \text{is-c-extralingual-character} \) of chapter 4.
7. INTRODUCTION TO THE INTERPRETATION OF ABSTRACT COMPILE TIME PROGRAMS

7.1 Introduction

Chapter 7 gives an informal treatment of the concepts used in the formal definition of the Interpreter in chapter 8. 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 6 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 definition of the CT-Interpreter.

In the following introduction little is said about the control component of the state of the CT-Machine, because the introduction does not deal with the various instruction schemata of the machine.

The prefix "ct-", denoting "compile-time", is omitted in this chapter wherever confusion cannot arise.

Explanations concerning the correspondence between concrete and abstract programs are enclosed in parenthesis, to indicate that this information is here only an additional help. Details can be found in chapters 3 to 6.

The adjectives "concrete" and "abstract" are only used when there is the danger of confusing concrete and abstract programs or parts of them.
7.1.1 Abstract compile time program

Abstract compile time programs are objects specified by the predicate is-ct-program (see chapter 6) and consist of two major parts, the declaration-part and the text-part-list.

ct-program:

- declaration-part
- text-part-list

The declaration-part is an object that associates identifiers with attributes. The association is realized by using each of the identifiers as a selector function which, if applied to the declaration part yields the corresponding attribute.

Such an object may be represented by a tree the branches of which are labeled by the mutually different identifiers, and the nodes hold the corresponding attributes. The left to right order of the branches is of course not significant.

Each of the attributes (to be described later) gives the associated identifier the properties either of a simple variable, of an entry-name, or of a label. (Each associated identifier attribute pair holds the static information that corresponds in a concrete ct-program either to a declaration of a variable or an entry name by means of a declare-statement, or to a procedure with or without a proper entry name declaration, or to a label to be found in the label-list of a ct-statement).

The text-part-list is a list of text-parts. A text-part consists of a ct-statement and of a ct-text.
(A text-part corresponds in a concrete ct-program to a ct-statement and the subsequent program text, in which all ct-procedures were replaced by the character "blank", up to the next ct-statement. If a ct-statement immediately is succeeded by a further ct-statement, the abstract ct-text is the empty list. If the concrete ct-program does not begin with a ct-statement, then the abstract ct-statement of the first text-part is an abstract null statement).

An abstract ct-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.

According to the Abstract Syntax (see chapter 6) an abstract ct-statement itself may contain a text-part-list with a structure as described above (resulting from the translation of concrete do-groups). This implies that lower level text-part-lists might be embedded in a higher level text-part-list up to any depth.

The intended semantics requires that a text-part-list has to be interpreted from left to right, i.e. its interpretation starts with its first text-part and continues with the second one, and so on, unless a text-part is reached, whose ct-statement changes this normal flow of control. Within a text-part at first the ct-statement is executed and then - if the normal flow of control is supposed - the associated ct-text is copied and the copy is scanned token by token, again from left to right. (A token is a syntactic unit that plays a role in the replacement process. Only a token which is an identifier possesses the syntactic properties that are a necessary condition for interpreting that identifier as a ct-variable or a ct-entry name). When a token is encountered during the scan denoting at that time a ct-variable or a ct-entry name, several tests have to be made concerning for instance the activation status, the existence of a value in the case of a ct-variable, or the number of actual parameters following the encountered token in the case of a ct-entry name. If all of those criteria are fulfilled, the token representing a ct-variable is replaced by its value, or in the case of a function reference the reference is replaced by the value the procedure returns. (Details, such as the scan of the single arguments of the function reference, the addition of blanks, etc. are discussed later). The substituted value itself now is handled like the copy of a ct-text and hence is rescanned from left to right and possible replacements are made. Each token of the copy that has passed this recursive sub-
stitution process, i.e. that is not further replaceable, belongs to the output of the macro processor and is concatenated with the character string composed of the former output-tokens. After the execution of the - in the dynamic sense - "last" text-part the so composed list of character values represents the entire output of the preprocessor.

A special abstract ct-statement may be mentioned here, namely the include statement, the execution of which gives access to a sequence of "external" abstract ct-programs via a list of address-pairs. (These external abstract ct-programs could be regarded as the output of the CT-Translator when applied to the concrete strings of external text incorporated by the corresponding concrete include statement). The intended semantics requires, that the declaration-part of the "main" program has to be modified by combining it with the declaration-part of the first incorporated external program. The effect of the include statement concerning the text-part-list of the first incorporated external program will be equivalent to the effect of inserting that text-part-list in place of the include statement. After the execution of the first text-part-list this process of combining declaration-parts and substituting a text-part-list and executing it has to be repeated until the whole sequence of external programs is executed.

7.1.2 General concepts of the interpretation

The semantics of the abstract compile time language, whose syntax is specified by the predicate is-ct-program, is defined by means of an abstract machine. The concept of the abstract CT-Machine is taken from /1/ and /6/. The reader should refer to these documents for a definition of the mechanism of an abstract machine.

The abstract machine is specified by the description of the of the states that the machine can assume and a state transition function. The states are considered to be abstract objects, their description is given by means of the abstract syntax in chapter 8,1. The transition function gives the rules for a computation of the machine as a finite sequence of states, and controls the transfer from one specific state to the next.

The substance of an initial state $S_0$ is the abstract main program which is to be interpreted and the possible set of external programs, elements of which could be invoked by include statements during the interpretation (see chapter 8,2).
7.2 The Declaration-Part

As stated in the introduction, the declaration-part associates identifiers with attributes. Each attribute defines the corresponding identifier to represent one of the following:

- a simple variable (in the following called "variable"),
- an entry name,
- or a label.

7.2.1 Variables

The attribute defining an identifier to represent a variable is either the elementary object described by the predicate is-INTG or the elementary object described by is-CHAR. (The ET-Translator gets the necessary information from the corresponding concrete declaration statement). In the first case the variable is predestined to hold as its value an optionally signed integer representation (only decimal integer arithmetic of precision \((n,o)\) is performed in the ET-facilities); in the second case a character string representation (ET-facilities handle only varying character strings that have no maximum length).

7.2.2 Labels

The attribute defining an identifier to represent a label is described by the predicate is-LAB. (The CT-Translator extracts this information from the labellists of the various concrete CT-statements contained in the program to be translated).

7.2.3 Entry-names

7.2.3.1 Structure of the associated attribute

An attribute defining an identifier to represent an entry-name has one of the following structures (also see 6-1(5)):

```plaintext
entry-decl
(1)
```

```plaintext
body
(2a)
```

```plaintext
identifier
(2b)
```
There are two reasons for the existence of five different possibilities: First, the reason for the distinction a - b follows from the fact that a ct-procedure may be associated with more than one entry-name. One of these entry-names holds the procedure-body (case a), the remaining entry-names hold this special entry-name instead of the body (case b). This indirect step from an entry-name via another entry-name to the corresponding procedure-body is of advantage for the Interpreter, because any procedure-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.

Secondly, the reason for 1, 2, and 3 is that not both the entry-declaration and the corresponding procedure-body have to appear in the program. The missing component may appear within the declaration-part of an external program which may be brought in by means of an include statement before an interpretation of the procedure can take place. Case 1 corresponds to the missing procedure-body, case 2 to the missing entry-declaration, in case 3 both entry-declaration and procedure are present.

An entry-declaration has the structure

(par-descr-list ret-type)

(In a concrete program it corresponds to the static information held by a declare statement declaring an entry-name. While a concrete declare statement occurring outside procedures possesses a dynamic property too - because of the fact that before the use of a variable or an entry-name the corresponding concrete declare statement must have been executed - and therefore is translated into an abstract declare statement which holds only the identifiers to be declared, a concrete declare statement occurring inside a procedure represents only a static information and hence is erased during translation).
7.2.3.2 The procedure-body

The procedure-body consists of four immediate components:

- A list of identifiers representing the formal parameters,
- the return type which is an elementary object described by
  \( \text{is-INTG or is-CHAR} \),
- the procedure-declaration-part,
- the procedure-text-part-list

(A procedure-body corresponds in a concrete ct-program to a ct-procedure without regard to the ct-entry-name-list. Similarly to the correspondence between concrete and abstract ct-programs the CT-Translator maps the concrete ct-p-sentence-list (see 3-7(34)) to the procedure-declaration-part and the procedure-text-part-list.)

The procedure-declaration-part differs from a declaration-part in that only variables and labels can be specified. (In a concrete program this corresponds to the fact that within a procedure neither entry-names can be declared nor procedures can be specified.)

The procedure-text-part-list has a structure similar to that of a text-part-list. But no ct-text to be scanned by the replacement mechanism exists here. In addition to that, the activate-, deactivate-, include-, and declare-statements are prohibited within a procedure-text-part-list. (Within a concrete procedure, declare statements may occur. Those are erased during the translation process). A further ct-statement is added, namely the return statement.

This gives rise to the definition of the following pairs of predicates in the Abstract Syntax (see chapter 6) indicating these differences:

- \( \text{is-ct-p-text-part} \)
- \( \text{is-ct-text-part} \)
- \( \text{is-ct-p-st} \)
- \( \text{is-ct-st} \)
- \( \text{is-ct-p-prop-st} \)
- \( \text{is-ct-prop-st} \)
- \( \text{is-ct-p-group} \)
- \( \text{is-ct-group} \)
- \( \text{is-ct-p-if-st} \)
- \( \text{is-ct-if-st} \)
where "p" stands for "procedure". To sum up, a procedure-text-part-list is a list of procedure-text-parts. A procedure-text-part is essentially a ct-statement which does not contain any ct-text. This statement may be a return-statement but may not be an activate-, deactivate-, include-, or declare statement.

7.3 Dynamic Properties of Identifiers and their Influence on the State

7.3.1 Scope of identifiers

The scope of an identifier declared in the declaration-part of the main program is the entire text-part-list of that program and the procedure-text-part-lists of those procedure-bodies that are contained in the declaration-part of that program and whose procedure-declaration-parts 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 procedure-text-part-lists of those procedure-bodies that are contained in the various declaration-parts of the external programs and whose procedure-declaration-parts do not redeclare that identifier belong to the scope of that identifier too.

The scope of an identifier declared in a procedure-declaration-part is limited to the corresponding procedure-text-part-list. (Note that only labels and variables can be declared within a procedure-declaration-part.)

The scope of an identifier declared in the declaration-part of an external program is the same as though this identifier were declared in the declaration-part of the main program. A redeclaration of an identifier by means of the declaration-part 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 the declaration-part of an external program, this declaration is not "known" until the first interpretation of this declaration-part.

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

7.3.1
7.3.2 Denotation of identifiers

In the case that an identifier is declared as a label the entire "denotation" of that identifier is defined to consist of an object described by the predicate is-LAB, i.e. the attribute the identifier is associated with in the declaration-part.

On the other hand, if an identifier is declared as a variable or as an entry-name, the associated attribute determines only a part of the "denotation". The rest of the denotation consists of dynamic components. Two of them which are necessary for variables as well as for entry names are discussed in the following.

Before a global variable or an entry-name is referred to in a ct-statement, a proper declare statement must have been executed. This one bit information, whether a proper declare statement has been executed up to that time or not, constitutes the "declare status" (T or F) as a component of the denotation. For the sake of uniformity local variables also possess this denotation component which is constantly T. Note that within a procedure-text-part-list no declare statements are allowed.

When a variable (necessarily global) or an entry-name is encountered during the scan of a ct-text, the question whether it is activated or deactivated is of importance. In the first case the substitution process becomes active, in the second case not. This one bit information (T or F) constitutes the "activate status" as a further component of the denotation. The activate status can be changed by the execution of an activate- or deactivate-statement, or by the first execution of a declare statement, which also activates an identifier.

7.3.3 The environment and the denotation directory

The association of an identifier with a denotation, initialized during the interpretation of the declaration-part of a program, holds during the interpretation of the whole text-part-list of the program, unless a procedure body becomes active whose procedure-declaration-part redeclares that identifier. In this case the old (global) denotation must be saved and the new one is initialized according to the procedure-declaration-part. Because of the scope rules (see 7.3.1) 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 attributes.

\[
E: \quad \text{id}_1 \ldots \text{id}_n \quad -\text{id}_i \text{identifiers} \\
\quad \text{ad}_1 \ldots \text{ad}_n \quad -\text{ad}_i \text{addresses}
\]

The denotation directory associates addresses with denotations

\[
\text{DN}: \quad \text{dn}_1 \ldots \text{dn}_n \quad -\text{dn}_i \text{denotations}
\]

This indirect step

\[
\text{id} \rightarrow E \rightarrow \text{ad} \rightarrow \text{DN} \rightarrow \text{dn}
\]

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 the same 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 (see 2-1(1)) from a pair consisting of the identifier for which the address is required and the scope information (global or name of the procedure). One hidden feature should be
noted at this point. An entry of the DN component of the state is never erased. The use of this one to one mapping ensures the "static" storage class for variables. (Every time a function call of one procedure is to be interpreted, which in consequence leads to the interpretation of the procedure-declaration-part, a new updating of E is performed. But the mapping for a locally defined identifier always specifies the same address. Therefore, a reference to the DN always gives the same entry and storage contents which could have been established by a previous execution of the same procedure.)

When control is transferred into a procedure-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 procedure-declaration-part of that body is established as the new environment. More exactly, each identifier declared in the procedure-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 procedure-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 procedure-declaration-part and is stored under the new address in the denotation directory.

When the procedure-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:

```
identifier
  E1  DN   address 1  denotation 1
  |     |     |     |     |     |
  E2  DN   address 2  denotation 2
```
Both addresses are connected in the denotation directory with a denotation for each, which are different in general.

7.3.4 Representation

Variables

Variables possess a further dynamic denotation component, the value representation, which is initialized by the first assignment to that variable and which can be changed by further assignments.

The situation is representable by the following diagram:

\[ \text{id} \rightarrow \text{ad} \rightarrow \text{DN} \rightarrow \text{ds} \rightarrow \text{vr} \]

\[ \text{E} \rightarrow \text{static} \]

\[ \text{vn} \rightarrow \text{dynamic} \]

where the abbreviations denote

E ... environment
DN ... denotation directory
id ... identifier
ad ... address
da ... data attribute
ds ... declare status
as ... activate status
vr ... value representation

In the case of a variable which is local to a procedure, the declare component of the denotation is static (T), and the activate status does not exist.

Entry-names

The introduction of an additional state-component, the "procedure-body directory" is necessary here. Each entry into this state component associates an address with a procedure body and its "execution status". The execution status holds one bit information (T or F) specifying whether the specific body is just being interpreted or not. With the aid of this component any attempt to reference a function recursively will be detected and marked as error.
In the following some examples are given, showing how the information which can be associated with entry-names is represented in the environment, denotation directory and procedure-body directory, according to the explanations given in 7.2.3.1.

a) Procedure-body with its three entry-names, where two of them are declared:

```
\[ \begin{align*}
\text{id}_1 & \rightarrow E \rightarrow \text{ad}_1 \rightarrow \text{DN} \\
\text{pd}_1, \text{rt} & \rightarrow \\
\text{ds}_1 & \\
\text{as}_1 & \\
\text{ad}_1 & \\
\text{P} & \rightarrow \text{body} \\
\text{id}_2 & \rightarrow E \rightarrow \text{ad}_2 \rightarrow \text{DN} \\
\text{pd}_2, \text{rt} & \\
\text{ds}_2 & \\
\text{as}_2 & \\
\text{ad}_1 & \\
\text{P} & \rightarrow \text{es} \\
\text{id}_3 & \rightarrow E \rightarrow \text{ad}_3 \rightarrow \text{DN} \\
\text{pd}, \text{rt}, \text{ad} & \rightarrow \\
\text{ds}, \text{as} & \\
\text{ad}_1 &
\end{align*} \]
```

where the abbreviations denote:

- **P** ... procedure-body directory
- **es** ... execution status
- **pd** ... parameter description
- **rt** ... return type
- **body** ... procedure-body

In the corresponding declaration-part of the program the identifier \text{id}_1 was associated with an entry-declaration and with the body (case 3a), \text{id}_2 was associated with an entry-declaration and with \text{id}_1 (case 3b). \text{id}_3 was associated with \text{id}_1 (case 2b). \text{ad}_1 is the address associated with \text{id}_1 here.

b) Procedure-body with two not declared entry names:

```
\[ \begin{align*}
\text{id}_1 & \rightarrow E \rightarrow \text{ad}_1 \rightarrow \text{DN} \\
\text{P} & \rightarrow \text{body} \\
\text{id}_2 & \rightarrow E \rightarrow \text{ad}_2 \rightarrow \text{DN} \\
\text{P} & \rightarrow \text{es}
\end{align*} \]
```

The identifier \text{id}_1 was associated in the declaration-part with the body (case 2a). The identifier \text{id}_2 was associated with \text{id}_1 (case 2b).
c) Declared entry-name without corresponding procedure-body:

\[ \text{id} \rightarrow \text{E} \rightarrow \text{ad} \rightarrow \text{DN} \rightarrow \text{ds} \]

The identifier \text{id} was associated in the declaration part with an entry-declaration (case 1).

### Labels

\[ \text{id} \rightarrow \text{E} \rightarrow \text{ad} \rightarrow \text{DN} \rightarrow \text{LAB} \]

### 7.4 Interpretation

#### 7.4.1 Interpretation of the declaration-part

The interpretation of the abstract program, which is contained (together with possible external programs) in the initial state \( \emptyset_0 \), 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 \( \text{ct-af} \) (cf. 2-1(1)). Each such identifier together with its address constitutes an entry into the environment. Any attribute 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 attribute contains a procedure-body an entry is also made in the procedure-body directory under the same address (compare with section 7.3.4).

The declare status component as well as the activate status component are initialized with the elementary object \( F \). In the case of a variable the value representation is not initialized, i.e. is \( \emptyset \). The execution status of a procedure-body is initialized with \( F \).

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 \( E, \text{DN}, \text{and P} \), the declaration-part is of no further use and therefore is erased.
7.4.2 Interpretation of the text-part-list

4.2.1 The state components "text storage", "statement counter", and "control dump"

When the interpretation of the declaration-part is finished the text-part-list has to be interpreted. Before this is done the text-part-list is stored in a state component of its own, the "text storage" (T). A further state component, the "statement counter" (SC) is initialized with 0. Generally, SC holds an integer value. This number is the element number of the just interpreted text-part in the text-part-list stored in T. Thus, at the time the next text-part is to be interpreted the statement counter is increased by 1.

According to the Abstract Syntax, a ct-statement can consist of or can contain a text-part-list again. Furthermore, the execution of an include statement causes the interpretation of at least one whole program. Hence, when control is transferred from a text-part-list into a lower level text-part-list the contents of T and SC - i.e. the former text-part-list and the element number of the just interpreted text-part, respectively - must be saved before T and SC are initialized with the adequate lower level data, in order to be able to continue the interpretation of the former text-part-list when the interpretation of the lower level text-part-list is finished. (It should be mentioned that the "control" component C of the state is treated here by the interpreter in analogy to T or SC.) This saving is realized by storing the contents of T and SC into a further state component, the "control dump" (CD). In the case that a previous control dump existed, its contents is also stored in the new control dump ("push down"-operation).

After the execution of any text-part it is tested whether the contents of SC is less than the length of the text-part-list stored in T or not. In the first case the contents of SC is augmented by 1 and the "next" text-part, i.e. the text-part out of T to which SC now points is interpreted. In the second case the interpretation of the current text-part-list is finished. The contents of T, SC, and CD are replaced by the T-, SC-, and CD-component, respectively, of the current control dump ("pop-up"-operation). After that the interpretation of the higher level text-part-list is continued either with the further interpretation of the ct-statement to which CS points (in the case of a group), or, if the interpretation of the ct-statement was finished, with the interpretation of the associated ct-text. (It is exactly that ct-text whose associated ct-statement caused the transfer of control into the lower level text-part-list).
Flow of control in the case that the ct-statement of the 3rd text-part is itself a text-part-list.

If the interpretation of a goto-statement leads to a transfer of control into a text-part-list lying more than one level deeper, then the push-down mechanism described above has to be applied for each single level (details are discussed in 7.4.3.2).

As pointed out in 7.1.2, the interpretation of a text-part comprehends the interpretation of its ct-statement and then the application of the scan- and replacement mechanism to the ct-text. In the following, the interpretation of the various ct-statements is discussed, thereafter the interpretation of the ct-text is explained in section 7.4.4.

7.4.2.2 Interpretation of the include statement

An include statement specifies a list of identifier-pairs. Each pair corresponds to an external program in the state component "external text storage" (ET). 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 ET yields the associated external program that has to be interpreted.

When a program is incorporated, a new level of the control dump CD is established, i.e. the contents of T, SC and CD are stored into CD, the text-part-list of the program is stored into T, and SC is initialized with its initial value O.

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 concerning all declaration parts interpreted up to now. Because of the global scope of identifiers declared outside of procedures together with the fact that multiple declarations are forbidden, 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 an attribute in the declaration-part, as described in 7.4.1.

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 only legal if the corresponding declaration-part is the null object, i.e. is $\emptyset$. Otherwise, the multiple declarations check described above leads to an error.

7.4.2.3 Interpretation of the goto statement

A goto statement interrupts the sequential flow of control and transfers it to a location marked by the label specified in the goto statement. It is not allowed to transfer the control by means of a goto statement out of a procedure or into a group (corresponding to a concrete do-group with iteration) or into an external program that is not under execution at that time (note that during the execution
of an included program further include statements may be executed. A jump from the lower level program back into the higher level program is allowed, independent of whether the higher level program is an external program or the main program). If there is no conflict with the above conditions, a goto statement may transfer the control out of a group, out of an external program, or out of or into a text-part-list or an if statement.

Roughly speaking, the interpretation of a goto statement is performed in two steps. In the first step the labeled statement to which control has to be transferred is searched for. In the second step control is transferred to that statement by performing suitable modifications to the state in order that the interpretation may continue with that statement and that the proper further flow of control is secured.

More precisely, the interpretation of a goto statement begins with a check whether the identifier contained in the goto statement has been declared as a label, i.e., the following must hold:

\[ \text{id \rightarrow ad \rightarrow DN \rightarrow LAB} \]

If that is the case, the current contents of the text-storage \( T \), i.e., the current text-part-list an element of which represents the goto statement under consideration, is scanned, element by element, and for each element it is examined whether the label-list of the ct-statement contains the identifier specified by the goto statement and if that is not the case, whether the ct-statement itself is a text-part-list or an if-statement. In this case the scan is continued one level lower, i.e., in the case of an if-statement the then- and else-components and in the case of a text-part-list its components are scanned in the same way. If this scan was successful, i.e., a ct-statement marked by the specified label was found, it yields as result a list of integers, called the "statement-counter-list" which specifies the location of the statement discovered within the text-part-list stored in \( T \). The first integer of the statement-counter list points to that element of \( T \) in which the statement in question was found. The second element indicates that the ct-statement specified by the first element is either an if-statement or a text-part-list which contains the statement searched for on a place specified by the second integer, and so on.

The following example shall illustrate the case, when a goto statement leads into a structure of embedded text-part-lists:
The corresponding statement-counter-list is <2, 4, 3>

There is one important case to be mentioned, namely the case, when the entire text-part-list stored in T had been scanned and the label specified by the goto statement was not found. A case distinction has to be made here:

1) The control dump component CD of the state is not \textcircled{Q}. This happens if the goto statement leads outside of a group, an external program, or a (procedure-) text-part-list. The text storage of the top element of the control dump is reinstalled as the current text storage and the control dump is popped up. The scan as described above is now repeated for the new text-part-list stored in T which is one level higher than the erased one.

2) The control dump is \textcircled{Q}. Then the goto statement is in error. It indicates that the outermost text-part-list of the main program itself or the outermost procedure-text-part-list of a body was scanned and the label was not found. (Note that during the interpretation of a function reference most of the former state components are stored into the "dump" D and that the new control dump CD is set to \textcircled{Q}.)

Suppose control has reached the location indicated by the goto statement, then the further flow of control - provided that no goto statement will be reached - goes towards the end of the text-part-list which contains the labeled ct-statement as an element. When the end of this text-part-list is reached, the next higher text-part-list is interpreted beginning at the point immediately after the ct-statement representing this lower level text-part-list. (Compare this with 7.4.2.1, concerning the end of a text-part-list.)
The same example as above may serve here as an illustration for the flow of control (note that any ct-text associated with a goto statement is never interpreted).

It is quite obvious that the information of levels 1 and 2 must be saved before the interpretation is continued on level 3.

Hence, the second step mentioned above is settled in the following way: When the label was found which yielded a statement-counter-list, the contents of the statement counter SC (which still points to the goto statement) is replaced by the first element of the statement-counter-list. In the case that control must lead into a lower level text-part-list (as in the example above) the current contents of the text storage T, the statement-counter SC and the control dump CD are combined and assigned to CD. Coincident with that, the next lower text-part-list to which SC points in T is installed in T and the corresponding element of the statement-counter-list is installed in SC. For each step deeper into the text-part-list structure this "push-down" of T and SC in CD and the installation of T and SC with the corresponding lower level information is repeated. When the label is reached the normal flow of control is continued.

Summary of the goto restrictions and how they are taken account of in the interpreter:

A goto statement must lead neither into a procedure-body nor into an external program that is not under interpretation; Obviously, the only possibility of entering a procedure-body is via the interpretation of a function reference. Note that bodies are not part of a text-part-list. They are stored in P. An external program which is not under interpretation can only be entered if it is first incorporated from the external text-storage ET and this can only be done by means of an include statement.
A goto statement must not lead into a group (corresponding to the concrete do-group with iteration). This is accounted for by the fact, that during the search for the specified label the components of a group are not scanned.

A goto statement must not lead out of a procedure: When a body is entered the contents of CD is dumped in D and CD is set to $\emptyset$.

A goto statement must not point to a declare statement: This case is settled by the translator which is responsible for the fact that abstract declare statements possess a label-list which is empty.

7.4.2.4 Interpretation of the group

A group corresponds to a concrete do-group with iteration. (As specified in the Concrete Syntax two kinds of ct-do-groups are allowed; one with iteration specification, the other without. The abstract form of ct-do-groups without iteration is described by the predicates is-ct-text-part-list and is-ct-p-text-part-list for groups outside or inside procedures respectively. Because of the special conditions for the use of labels inside of ct-do-groups with iteration they are separated from ct-do-groups without iteration and are represented by the predicates is-ct-group and is-ct-p-group). According to the Abstract Syntax, a group consists of an iteration and a text-part-list. The iteration consists of a variable, the initialization-expression, the by-expression, and the to-expression. The by-component as well as the to-component may also be $\emptyset$, but not the by-component alone. (This happens if the corresponding by- and/or to-component in the concrete do-group was missing. If only the by-component was missing, a representation of the value 1 was substituted instead of $\emptyset$.)

At the very beginning of the interpretation it is checked whether the identifier representing the iteration variable possesses a proper denotation, i.e. whether the corresponding component of the denotation directory DN has an attribute component that is INTG or CHAR, and a declare component that is T.

If this is the case the initialization-expression is evaluated and assigned to the iteration variable (possible conversions included). Now, three case distinctions have to be made:

1) The by-component as well as the to-component of the iteration are expressions.
2) The by-component is an expression, but the to-component is $\emptyset$
3) Both by- and to-components are $\emptyset$
In the first case both expressions are to be evaluated. In spite of the iterated execution of the text-part-list of the group these expressions are evaluated only once. Now, an iteration will take place if the end-condition of the iteration process is not met. The interpretation of one iteration of the group comprehends the following actions: The text-part-list of the group is to be interpreted. As already mentioned above, at first the contents of the state components T (representing the text-part-list containing also the group under considerations) and SC (pointing to the group) are dumped in the control dump. T and SC are initialized with the text-part-list of the group and with 0, respectively. When the interpretation of the text-part-list is finished, the dumped T- and SC-components are reinstalled, the value of the iteration variable will be updated by adding to it the value of the by-expression, and the end-condition is checked again to decide whether a further loop is necessary.

In the second case, i.e. only the to-component is \( \Omega \) (corresponding to a concrete do-group without specification of the to-expression), the iteration continues over and over again, unless the interpretation of a goto-statement leads out of the text-part-list of the group.

In the third case (in the concrete group neither the by- nor the to-expressions were specified), the text-part-list of the group is interpreted one and only one time.

### 7.4.2.5 Interpretation of the if-statement

The abstract if-statement contains a decision component which is an expression, and the then- and else-component, each representing a single ct-statement. (If there was in the concrete if-statement no else-part specified, the Translator inserts a null statement in the place of the missing else-part).

The interpretation starts with the evaluation of the specified expression. The value is then converted to a bit-string, if necessary. If any bit in the resulting string represents the value 1, the then-component is interpreted, and the interpretation of the if-statement is finished. If all bits represent the value 0, the else-component is interpreted.

Transfer of control by means of a goto statement into or out of an if-statement is allowed.
7.4.2.6 Interpretation of the declare statement, the activate-, and the deactivate statement

a) Declare statement

The abstract declare statement contains only an identifier-list. It may occur within a text-part-list, but not in a procedure body, i.e., not in a procedure-text-part-list. When a declare statement is interpreted the first time, the denotation of each of the component identifiers is modified by changing the declare-component and the activate-component of the associated denotation from F to T. When control reaches the declare statement at least the second time, indicated by the fact that the declare-component of the denotation of each specified identifier is T, the interpretation is in analogy with that of a null statement. Identifiers occurring in a declare statement denote either a variable or an entry-name.

(Concrete declare statements may also occur within procedures. The Translator extracts the static information contained in concrete declare statements and collects it in the declaration-part of the abstract program. What remains, is the "dynamic" property of concrete declare statements not occurring in procedures, namely that before a global variable or entry-name can be used within a ct-statement, a corresponding declare statement must have been executed. Furthermore, the first execution of a declare statement not occurring in a procedure, also activates the specified identifiers. This is the reason for the existence of abstract declare statements containing only an identifier-list. Because of the fact that - in the concrete case - declare statements within procedures can be placed elsewhere and the activation of a local variable is ruled out, there is no necessity for abstract declare statements within bodies, i.e., within procedure-text-part-lists).

b) Activate- and deactivate statement

These two statements control the activation status of the ct-variables and entry-names, which influences the execution of the ct-text scan mechanism by marking identifiers as replaceable or not.

Activate statements as well as deactivate statements must not occur within procedure-text-part-lists. Before such a statement is executed, each identifier of the specified identifier-list must have been a member of an interpreted declare statement, i.e., the declare component of the denotation of each identifier must be "T". There is only one exception: The built-in function SUBSTR cannot be declared explicitly, but may be activated and deactivated (see 7.4.3.3).
With the exception of the SUBSTR-case, the interpretation of an activate statement consists of setting the activate component of the denotation of each specified identifier to "T". The interpretation of a deactivate statement consists of setting the activate component of the denotation of each specified identifier to "F".

c) Representation

Let "decl-st" stand for "declare-status" and "act-st" for "activate status". These two components of the denotation of an identifier, representing an entry-name or a global variable, are illustrated by an example starting with their initialization:

\[
\begin{align*}
\text{interpretation of a declaration-part} & \\
\text{decl-st:} & F \\
\text{act-st:} & F \\
\text{first interpretation of the corresponding declare statement} & \\
\text{decl-st:} & T \\
\text{act-st:} & T \\
\text{interpretation of a corresponding deactivate statement} & \\
\text{decl-st:} & T \\
\text{act-st:} & F \\
\text{interpretation of a corresponding activate statement} & \\
\text{decl-st:} & T \\
\text{act-st:} & T
\end{align*}
\]

\[\text{decl-st:} T \ldots \text{necessary condition for using the corresponding identifier in a ct-statement, except of the declare statement.}\]

\[\text{decl-st:} T \ldots \text{necessary condition for a reference of the corresponding identifier act-st:} T \text{ from a ct-text, i.e., for replacement.}\]
Note that a variable which is local to a procedure, i.e., which is declared in a procedure-declaration-part, always has (starting with the interpretation of the procedure-decl-part):

\[ \text{decl-st: T} \]
\[ \text{act-st: } \square \]

### 7.4.2.7 Interpretation of the assignment statement

An abstract assignment statement contains a left part which is an identifier and a right part which is an expression. The left part must denote a declared variable, i.e., the attribute component of the corresponding entry in the denotation directory DN must be INTG or CHAR, and the declare-component must be T.

The expression will be evaluated and the result will be converted to meet the attribute requirements of the variable specified by the attribute component of its denotation. Then, the value representation of the converted result is assigned to the storage component of the denotation of the variable.

### 7.4.2.8 Evaluation of expressions

Expressions, objects satisfying is-ct-expr, are interpreted corresponding to their type. Because of the structure of the abstract object, precedence rules for the different kinds of operators are not necessary here, they are built into the form of the object immediately by the CT-Translator.

The case of a parenthesized expression (is-ct-paren-expr) is taken into the Abstract Syntax because of argument passing in function procedures. In such a case it is not the same whether a variable is surrounded by parenthesis or not (call by value or call by name).

The elementary operands of expressions are constants, variables, and function references. In general, variable- and function references are syntactically indistinguishable; the denotation of the corresponding identifier settles this matter. The result of the evaluation of a variable- or function reference is a constant (see next section).

Before the evaluation of an infix or prefix operation can take place, the evaluated operands to which the operation is to be applied have to be converted to the proper data type. The types of the evaluated operands and the operator of the
operation determine this target data type.

For the various kinds of conversions the conversion process allows by means of case distinctions the necessary check to be made on the source object. For example, during the character-string to integer conversion, the value of the character string has to have the form of a possibly signed sequence of digits, optionally surrounded by blank characters to allow the conversion to an integer value. This value then has to be checked whether, because of the implementation defined precision, a truncation has to take place. The exact form of this truncation depends on the specific implementation and is represented by means of the implementation defined function ct-truncate.

After the conversion the execution has to be done for the various types of operators.

The result of a comparison operation is a bit string of length 1 with the value 1 for true and 0 for false. Character strings have to be compared according to an implementation defined collating sequence represented by means of the implementation defined function ct-collate, mapping characters into integers. Therefore the comparison can be done on these resulting arithmetic values.

For the arithmetic operations the resulting value has to be checked whether it has to be truncated. Note that the result always is an integer value even in the case of a division.

7.4.3 Evaluation of references appearing in expressions

7.4.3.1 Reference of a variable

A reference to a global variable is allowed only after the corresponding declare statement has been interpreted. This interpretation is indicated by the declare component of the corresponding entry in the denotation directory. Variables that are local to a procedure have no associated declare statement. In order to avoid any difference in the treatment of references between these two sorts of variables, the declare component of the denotation of a local variable is initialized with "T" during the interpretation of the procedure-declaration-part in which the local variable is declared.

The result of interpreting a reference is an abstract object with the form of a ct-constant. It has two components, the data-attribute that is stored in the attribute
bute component of the denotation, and the value of the variable in an implementation defined representation which is found in the storage component of the variable denotation. If this component does not exist an error is encountered.

7.4.3.2 Reference of a function procedure

**Necessary checks:**

A reference is allowed only if the corresponding declare statement has been interpreted, which is indicated by a "T" in the declare component of the denotation. It is also necessary that the storage component of the corresponding denotation exists, which gives access to the proper entry in the procedure body directory where the procedure body is stored, otherwise an error occurs. (The storage component of the denotation could be missing, because of the possibility of the separate declaration of entries and their associated procedure-body within different declaration-parts). Before the interpretation of the function reference can take place, it is necessary to examine whether this execution will lead to a recursive use of the procedure body. The check is done on the "execution status" component of the procedure-body entry in P. If this component is T, it indicates that a previous function call still is not completed and a new reference is forbidden.

The following is an illustration of the conditions above:

```
 attribute ... E \rightarrow ad_1 \rightarrow DN
 id \rightarrow declare status : T
 activate status : ... 
 body component : ... 
 storage : ad_2 \rightarrow p \rightarrow execution status : F
```

A further test is, whether the length of the argument-list of the reference is equal to the length of the parameter description list specified in the attribute component of the denotation.

**The interpretation of function arguments:**

The arguments of a function reference are to be evaluated from left to right. For the evaluation of one argument out of the argument list, which is an expression, the corresponding element of the parameter description list, stored in the attribute component of the denotation of the entry name, is necessary.
For the evaluation of an argument two distinct cases can occur:

(a) If the argument is a variable, a check is made whether its attribute, found in the corresponding entry of DN, matches that given by the associated parameter descriptor, or whether the parameter descriptor is "*" (i.e. was not specified in the concrete entry-name declaration). In both cases the address of the variable is to be passed to the corresponding parameter of the procedure. In turn any use of those parameters will result in a reference of the DN entry of the argument, e.g., the storage component of the argument may be changed inside of the procedure by means of an assignment statement ("call by name").

(b) In all other cases the argument is evaluated, converted according to the corresponding parameter descriptor, and the creation of a dummy entry in DN will take place. (There is one exception: the evaluated argument represents a bit string and the parameter descriptor is "*". This case leads to an error). The denotation of the dummy holds the value representation in the storage component, the data attribute in the attribute component, and "T" in the declare component. This concept allows the same handling routine for both types of argument-passing during their connection with the corresponding parameters of the referenced function procedure.

Installation of the procedure-body, the state components "dump" D and "return information" RI:

During the interpretation of a procedure-body some components of the state are used to hold the necessary local information, which is of no further use when the interpretation of the function reference is finished. When this is the case, the former contents of the state components must be reinstalled in order to secure proper continuation of the interpretation.

Hence, before a procedure-body is entered, the contents of these state components are saved in the dump. Then these state components are initialized as follows:

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

The text storage \(T\) holds the procedure-text-part-list of the procedure-body
and the statement counter \(SC\) is set to start the interpretation with the first
list element.

The control dump \(CD\) is set to \(\emptyset\) to indicate that \(T\) holds a procedure text-
part-list. This information is used during the interpretation of goto statements.

The state component "return information" \(RI\) holds data that is necessary
for the interpretation of a return statement and which is available only at the
beginning of the interpretation of the function reference. \(RI\) consists of three
components:

The return type component holds the type to which the result is to be con-
verted before it is assigned to a dummy which is specified in the un-name
component. This type is copied out of the attribute component of the deno-
tation of the entry-name.

The pointer component holds the address under which the procedure body and
its execution status can be found in \(P\). The address is copied out of the
storage component of the denotation of the entry-name.

The un-name component specifies a unique name which has been generated when
the function reference was encountered.

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 cor-
responding entry in the procedure-body directory \(P\) is set to "T".

Interpretation of the procedure body:

A number of checks have to be done now:

whether all elements of the parameter list of the body are declared
in the procedure-declaration-part,
whether all elements of the parameter list are mutually different,

whether the attribute which is associated in the procedure-decl-part with any formal parameter out of the parameter list matches the attribute of the corresponding passed argument which is found in the attribute component of the proper DN entry,

whether the length of the parameter list is equal to the length of the passed argument list,

whether the return type specified in the body is the same as the return type specified in the attribute component of the denotation of the entry-name (or the return type specified in the state component RI).

If all checks are positive - the contrary would cause an error - the environment is updated by all identifiers declared in the procedure-declaration-part which are not used as parameters. If a redeclaration of an identifier occurs, the existing old entry in E is overriden. The Abstract Syntax ensures that only variable and label declarations can appear in the procedure-declaration-part.

After that, all necessary entries for the new declared identifiers into the DN component of the state are made. In the case of a label only the attribute component of the corresponding denotation is initialized with "LAB". In the case of a variable the attribute component of the corresponding denotation is assigned with the corresponding attribute of the procedure-decl-part (INTG or CHAR), and the declare component gets a "T". Note that by the definition of the address function ct-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.

Also, the connection of the parameters with their corresponding arguments will be performed now. The arguments have been interpreted at the time the function reference had been encountered and for each argument an address was passed, indicating its entry in DN. This address now is connected with the corresponding parameter in an entry into the environment E.

Interpretation of the procedure-text-part-list:

The interpretation of a procedure-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 ct-statement and a ct-text that has to be scanned for possible replacements when the interpretation of the ct-statement is finished, a procedure-text-part, on the other hand, only consists of a ct-statement which itself must not con-
tain any ct-text. Hence the interpretation of a procedure-text-part consists only of the interpretation of a ct-statement.

It should be noted that the Abstract Syntax does not allow

- activate statements
- deactivate statements
- include statements
- declare statements

inside a procedure-text-part-list. (In the concrete case declare statements are not forbidden.)

Interpretation of 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 procedure-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 data attribute stored in the return-type component of RI 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 un-name component of RI, i.e., an entry for the dummy in DN is made, with the value representation stored in the storage component and the data attribute stored in the attribute component. The address found in the pointer 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 of the procedure block, because the unique name was saved here too (in the control). This constant is restored, and the interpretation of the expression that was interrupted by the function reference continues with the constant in place of the reference.
7.4.3.3 Reference of the built-in function SUBSTR

The unique compile time built-in function SUBSTR cannot be declared explicitly. On the other hand, the name SUBSTR is not a reserved word and therefore could be used under certain circumstances also as an entry-name of a user-defined procedure, or as a variable or label.

When during the interpretation of a ct-program the identifier SUBSTR is encountered for the first time and it occurred:

1. either in an activate or deactivate statement or in a ct-expression as a reference, even if that expression appears inside a procedure-body, then this and any further occurrence of SUBSTR is defined to refer to the built-in function, except for occurrences that appear within procedure-bodies that re-declare the identifier SUBSTR. If in the further flow of interpretation an external program is incorporated by means of an include statement and the declaration-part of that program contains a declaration for SUBSTR, then a multiple declarations error occurs.

2. in the declaration-part of the main program or in the declaration-part of an incorporated external program, then the further interpretation acts, as if no built-in function was defined for the compile-time facilities.

3. in a procedure-declaration-part, then SUBSTR has within that procedure the specified meaning, outside that procedure the meaning of SUBSTR is left open until during the further interpretation case (1) or (2) is reached.

4. within a ct-text, then this token is copied into the result component of the state, because it was not activated. The meaning of SUBSTR is left open as in (3).

5. within a context not covered by the points 1 - 4, then an error occurs.

These conditions are taken into account in the Interpreter in the following way: When SUBSTR is encountered in an activate- or deactivate statement or within an expression it is tested whether that is the dynamically first occurrence of SUBSTR (in the sense of case (1), with possible precedings of cases (3) and (4)). In the case of the activate- or deactivate statement it is checked whether SUBSTR has no entry in the current environment E. In the case of an expression it is checked whether SUBSTR has no entry in the current environment as well as in the outermost environment which is stored in the bottom of the dump D. (The reason for the ad-
ditional check in the second case is that the expression may belong to a procedure). If this test indicates that SUBSTR is not "known", an entry for SUBSTR must be made in the outermost environment even if the reference to SUBSTR appears inside of a procedure. The address which is now associated with SUBSTR in the outermost environment is connected with a special denotation in DN, that gives SUBSTR the meaning of the built-in function.

```
SUBSTR --> E_O --> ad --> DN
attribute : BUILTIN
declare status : T
activate status : F
```

$E_O$, ..., outermost environment

For the further interpretation the effect is as if the identifier SUBSTR had been declared in the declaration-part as built-in function. Now, the normal interpretation of the activate- or deactivate statement, or of the reference starts.

Proper interpretation of a reference to the built-in function SUBSTR:

The entire check that has to be done when SUBSTR is encountered within an expression, to decide whether SUBSTR refers to the built-in function, is represented here by means of a table:

<table>
<thead>
<tr>
<th>attribute component of the denotation of SUBSTR via the outermost environment</th>
<th>current environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) not defined</td>
<td>not defined</td>
</tr>
<tr>
<td>b) BUILTIN</td>
<td>not defined</td>
</tr>
<tr>
<td>c) BUILTIN</td>
<td>BUILTIN</td>
</tr>
<tr>
<td>d) BUILTIN</td>
<td>other attribute</td>
</tr>
<tr>
<td>e) not defined</td>
<td>other attribute</td>
</tr>
<tr>
<td>f) other attribute</td>
<td>other attribute</td>
</tr>
</tbody>
</table>

"not defined" indicates that SUBSTR has no entry in the environment under consideration, thus no denotation of SUBSTR exists via that environment.

The cases a), b), and c) refer to the built-in function. Case a) is equivalent to the case described above, where the proper denotation was assigned to SUBSTR. Case b) comes true during the interpretation of a procedure-text-part-list, if during the same interpretation of the procedure a former occurrence of SUBSTR had fullfill-
ed case a). Case c) happens within a procedure, whose invocation occurred at a time
where a case a) or b) had already been interpreted. (It is obvious that in case a)
and c) the outermost environment coincides with the current environment if the
reference to SUBSTR does not occur within a procedure.)

The cases d), e), and f) do not refer to the built-in function. d) is the case
when outside a procedure SUBSTR has the denotation of the built-in function, but
SUBSTR is referenced within a procedure that proclaims SUBSTR. Case e) follows from
point 3) above. Case f) follows from point 2).

After SUBSTR had been identified as built-in, the specified 2 or 3 arguments
are evaluated and converted (if necessary) according to the data attributes CHAR,
INTC, INTG, respectively.

Out of the first argument a substring has to be built depending on the second
and third arguments. The second argument specifies the starting-position for the
substring. The third argument, if given, specifies its length. The omission of the
third argument has the meaning that starting from the position given by the second
argument the whole string has to be taken. The function returns a ct-constant of
char-string type.

7.4.4 The scan and replacement mechanism for the ct-text

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

The interpretation of a ct-text starts by copying it out of the corresponding
text-part, whereby the character value BLANK added as first element to the copy en-

sures that in the generated output of the Interpreter a concrete character blank
occurs where in the corresponding input a ct-statement had occurred.

Roughly speaking, the interpretation of the copied ct-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 then the next token is produced.

This output medium is represented by the state component \( R \) (result). "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 source-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/, /6/).

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 is known, but the attribute component of its denotation does not exist (procedure without entry-name declaration).
3. The corresponding abstract identifier denotes a label.
4. The activation component of the denotation of the corresponding abstract identifier is "F" (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 the activated built-in function.

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 storage component of the denotation is \( \emptyset \), 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 of a variable, of a function procedure, or of the built-in function SUBSTR was successful, i.e. a ct-constant was returned, this constant is at first converted to a character string, if necessary. Then this character string, whose elements are represented in an implementation defined manner, is transformed into the corresponding list of character values. 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. This list is not yet transferred to R instead of the corresponding reference, but rather is immediately rescanned for further replacement, as if it were the copy of a ct-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 storage component of the corresponding denotation exists), a check has to be made whether the procedure possesses parameters. If it is the case (indicated by the attribute component of the corresponding denotation), the list of character values following that entry-name 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 description list found in the attribute component of the corresponding denotation, 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 ct-text, with the only difference, that any token that is encountered during this interpretation and which is not (further) replaceable, is not transferred to R, but is transferred into the storage component of a dummy, which was generated when the scan and replacement mechanism started for the argument under consideration. (Note that care is taken that this position contains a proper internal value representation for the char-val-list). After the rescan is completed for the argument, the character string contained in the storage component of 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 7.4.3.2.

For the built-in function the same mechanism as above is applied to its arguments, before the interpretation of the built-in function can take place.

### 7.5 Summary of the State Components and their Properties

Abbreviations for the major components of the state $\Psi$ are introduced which will be used in the formal definition (chapter 8). The terms given, name the major state parts according to their contents and use.

<table>
<thead>
<tr>
<th>Component</th>
<th>Abbreviation</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>s-ct-et($)</td>
<td>ctET</td>
<td>External Text Storage</td>
</tr>
<tr>
<td>s-ct-r($)</td>
<td>ctR</td>
<td>Result Cell</td>
</tr>
<tr>
<td>s-ct-un($)</td>
<td>ctUN</td>
<td>Unique Name Counter</td>
</tr>
<tr>
<td>s-ct-e($)</td>
<td>ctE</td>
<td>Environment</td>
</tr>
<tr>
<td>s-ct-dn($)</td>
<td>ctDN</td>
<td>Denotation Directory</td>
</tr>
<tr>
<td>s-ct-p($)</td>
<td>ctP</td>
<td>Procedure Body Directory</td>
</tr>
<tr>
<td>s-ct-t($)</td>
<td>ctT</td>
<td>Text Storage</td>
</tr>
<tr>
<td>s-ct-sc($)</td>
<td>ctSC</td>
<td>Statement Counter</td>
</tr>
<tr>
<td>s-ct-c($)</td>
<td>ctC</td>
<td>Control</td>
</tr>
<tr>
<td>s-ct-dn($)</td>
<td>ctCD</td>
<td>Control Dump</td>
</tr>
<tr>
<td>s-ct-ri($)</td>
<td>ctRI</td>
<td>Return Information</td>
</tr>
<tr>
<td>s-ct-d($)</td>
<td>ctD</td>
<td>Dump</td>
</tr>
</tbody>
</table>

#### 7.5.1 The external text storage ctET

The external text storage ctET 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-ct-program. (These external programs could be regarded as the output of the CT-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 also may be a "*". The corresponding external program is contained in the external text storage under the selector ct-sel(idp).
Note that ctET is the unique state component that remains unchanged during the entire interpretation process.

7.5.2 The result cell ctR

During the interpretation of a program the ct-text is scanned and possible replacements are made. The result is copied in an output medium of the machine. This output medium is represented by the state component ctR. When the interpretation of the compile time program is terminated, the list of character values specified by ctR is considered to be a concrete PL/I program which can be the input to the parsing-translation-interpretation process of PL/I [4], [6].

7.5.3 The unique name counter ctUN

In two situations during the interpretation of a ct-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 ct-text.

The unique name counter ctUN 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 ct-un-name returns one which is different from all unique names used before.

Note, that the ranges of ct-un-name and ct-af have no elements in common.

7.5.4 The environment ctE

The environment associates identifiers occurring in a declaration-part or in a procedure-declaration-part with addresses under which the corresponding denotation is found in the denotation directory. The addresses are generated by the one-to-one function ct-af, mapping scope information ("*" for global scope, or the entry-name or the address of a procedure-body) and an identifier (in the case of a parameter
passing an integer value) 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 procedure-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 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 procedure-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.

7.5.5 The denotation directory ctdN

The denotation directory associates addresses (is-ad) and unique names (is-n) with denotations. In general, a denotation consists of four components. They are distinguished by the selectors:

s-ct-at gives access to the type description of the corresponding entry;

s-ct-dcl holds the declare status, indicating whether a proper declare statement has been interpreted or not;

s-ct-act holds the activate status of the corresponding identifier. The component is changed by the interpretation of a declare, activate or deactivate statement;

s-ct-s specifies the storage component of the denotation. For variables it holds the values of the variables in the implement-
tation defined internal value representation which is defined by the function ct-represent given in chapter 2. For entry-names the storage contains an address which gives access to the body of the specified procedure. This body is found in the state component ctP. Labels and the built-in function do not have a s-ct-s component in their denotation.

7.5.6 The procedure body directory ctP

The component ctP 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 storage component s-ct-s of the denotation of the specified entry-name.

s-ct-body selects the procedure body;

s-ct-cld 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.

7.5.7 The text storage ctT

The text storage contains the text-part-list or procedure-text-part-list which is currently being interpreted. Together with the statement counter ctSC it gives access to the point where the interpretation is taking place, so that the next text-part or procedure-text-part can be found.

When during the interpretation of a (procedure-) text-part-list a lower level (procedure-) text-part-list is reached, the first one, i.e., the current contents of ctT, will be saved in the s-ct-t component of the control dump ctCD and the new (procedure-) text-part-list is assigned to ctT. When the interpretation of the current contents of ctT is finished the s-ct-t component of ctCD is reinstalled in ctT and the interpretation continues on the higher level.
7.5.8 The statement counter ctSC

The statement counter holds an integer value. This number is the element number of the (procedure-) text-part which is currently under interpretation out of the (procedure-) text-part-list stored in ctT. ctSC will be updated by 1 each time a new (procedure-) text-part is to be interpreted.

ctSC must be saved and reinstalled concurrently with the ctT state component.

7.5.9 The control ctC

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 candidates for immediate execution. For more explicit information about the control and its cooperation with the state transition function refer to /1/ and /6/.

7.5.10 The control dump ctCD

The control dump represents one of the two push down mechanisms of the CT-Machine. The ctCD holds the information which is necessary to resume the further interpretation, after the interpretation of a group, a lower level text-part-list or an incorporated external program is finished, ctSC is also to be used for the proper interpretation of goto statements.

The control dump will be set to $@$ at the time the interpretation of a function reference starts, to forbid goto statements leading out of the procedure-body. There are four components of the control dump:

s-ct-t holds the old text-storage, i.e., that (procedure-) text-part-list to which control is to be transferred back when the interpretation of the current contents of ctT is finished.

s-ct-sc holds the old statement counter, i.e., the integer value pointing to that element of the s-ct-t component of ctCD which is just under interpretation.
s-ct-c  holds the old control which will resume the computation
after the current (procedure-) text-part-list interpretation
is finished.

s-ct-cd  holds previous levels of the control dump.

7.5.11 The return information ctRI

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 only available at the beginning of the interpretation of the
procedure. The state component ctRI will be used to save such information. It con­sists of three components:

s-ct-name  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-ct-ret-type  gives information about the type to which the result is to
be converted before it is assigned to the dummy name speci­fied above.

s-ct-pt  is used to identify the entry in ctD, so that its s-ct-cld 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.

7.5.12 The dump ctD

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 seven components of the ctD:

s-ct-e  holds the environment that was active when the function
reference was encountered.
s-ct-t holds the old \textit{ctT} component, i.e., the (procedure-) text-part-list that was under interpretation when the function reference was encountered.

s-ct-sc holds the old statement counter, i.e., an integer value pointing to that element of the s-ct-t component which contains the function reference.

s-ct-c saves the old control, because a new control must be established for the time of the interpretation of the procedure-body.

s-ct-cd saves the old control dump. The current control dump is set to $\varnothing$ to forbid \texttt{goto}'s leading outside the procedure-body.

s-ct-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-ct-d holds previous levels of the dump.
8. INTERPRETER

8.1 Abstract Syntax of the CT-Machine States

(1) \textit{is-ct-state} = (\langle \textit{is-ct-et} : \textit{is-ct-et} \rangle, \\
\langle \textit{is-ct-r} : \textit{is-char-val-list} \rangle, \\
\langle \textit{is-ct-un} : \textit{is-intg-val} \rangle, \\
\langle \textit{is-ct-e} : \textit{is-ct-e} \rangle, \\
\langle \textit{is-ct-dn} : \textit{is-ct-dn} \rangle, \\
\langle \textit{is-ct-p} : \textit{is-ct-p} \rangle, \\
\langle \textit{is-ct-t} : \textit{is-ct-t} \rangle, \\
\langle \textit{is-ct-sc} : \textit{is-intg-val} \rangle, \\
\langle \textit{is-ct-c} : \textit{is-ct-c} \rangle, \\
\langle \textit{is-ct-cd} : \textit{is-ct-cd} \rangle, \\
\langle \textit{is-ct-ri} : \textit{is-ct-ri} \rangle, \\
\langle \textit{is-ct-d} : \textit{is-ct-d} \rangle) \\
\text{Ref.: is-ct-c 7.5.9}

(2) \textit{is-ct-et} = \{ \langle \textit{ct-sel(idp)} : \textit{is-ct-program} \rangle \mid \langle \textit{is-ct-id-pair(idp)} \rangle \}
\text{Ref.: ct-sel 2-3(7)}

(3) \textit{is-ct-e} = \{ \langle \textit{id} : \textit{is-ad} \rangle \mid \langle \textit{id} \rangle \}

(4) \textit{is-ct-dn} = \{ \langle \textit{ad} : (\langle \textit{ct-at} : \textit{is-ct-prop-var} \rangle \mid \textit{is-ct-entry-decl} \rangle \rangle \mid \langle \textit{is-BUILTIN} \rangle \mid \langle \textit{is-LAB} \rangle \mid \langle \textit{is-ct} \rangle, \\
\langle \textit{ct-act} : \textit{T} \rangle \mid \langle \textit{is-F} \rangle \mid \langle \textit{is-Q} \rangle, \\
\langle \textit{ct-dcl} : \textit{T} \rangle \mid \langle \textit{is-F} \rangle \mid \langle \textit{is-Q} \rangle, \\
\langle \textit{ct-s} : \textit{is-ct-vr} \rangle \mid \langle \textit{is-ct-vr-list} \rangle \mid \langle \textit{is-ad} \rangle \mid \langle \textit{is-Q} \rangle \rangle \mid \langle \textit{is-ad(ad)} \rangle \mid \langle \textit{is-n(ad)} \rangle \}

(5) \textit{is-ct-p} = \{ \langle \textit{ad} : (\langle \textit{ct-body} : \textit{is-ct-body} \rangle, \\
\langle \textit{ct-cld} : \textit{T} \rangle \rangle \mid \langle \textit{is-ad(ad)} \rangle \}

\text{Ref.: is-ct-c 7.5.9}
8.2 The Initial State of the CT-Machine

The interpretation of a ct-program starts with an initial state $s_0$ of the CT-machine. This initial state contains the ct-program to be interpreted (the "main program") as argument of the initial instruction ct-program-text-execute, and a set of labeled external programs.

Hence, the initial state is a function depending on the following two parameters:

- $t_o$ the abstract ct-program to be interpreted. It satisfies the predicate is-ct-program defined by the Abstract Syntax in chapter 6.
- $et_o$ the external text storage described by the predicate is-ct-et (in 8.1). $et_o$ remains unchanged during the entire interpretation process.
8.3 Formal Definition of the CT-Facilities Semantics

8.3.1 Program execution and interpretation of the declaration-part

A program execution starts at the very beginning of a computation of the CT-machine, but also when an external program is incorporated by means of an include statement (cf. 8.3.9). In both cases the interpretation is done on an abstract object satisfying the predicate is-ct-program.

Metavariables

\[ \text{id} \quad \text{is-id} \]
\[ \text{ad} \quad \text{is-ad} \]

(12) \text{ct-program-text-execute}(t) =

\[ \text{s-ct-t}: \text{s-ct-text-part-list}(t) \]
\[ \text{s-ct-sc}: 0 \]
\[ \text{s-ct-cd}: \text{ct-stack-cd}(\xi) \]
\[ \text{s-ct-c}: \text{int-ct-next-text-part}; \]
\[ \text{int-ct-decl-part}(\text{s-ct-decl-part}(t)); \]
\[ \text{update-cte}(\text{s-ct-decl-part}(t)) \]

for: is-ct-program(t)

Ref.: \text{int-ct-next-text-part} 8-6(22)

Note: The new level of the control dump is necessary only when an external program is interpreted.

(13) \text{ct-stack-cd}(s) =

\[ \mu(t, \xi) \]
\[ \text{s-ct-t: s-ct-t}(s), \text{s-ct-sc: s-ct-sc}(s), \]
\[ \text{s-ct-cd: s-ct-cd}(s), \text{s-ct-c: s-ct-c}(s) \]

for: is-ct-state(s)
(14) \( \text{update-cte}(t) = \)
\[
(\forall \text{id}) (\text{is-ct-ident}(t) \supset \text{is-Q} \circ \text{id}(\text{ctE}) \lor \text{is-ct-entry} \circ \text{id}(t) \land \text{is-Q} \circ \text{s-ct-entry-decl} \circ \text{id}(t) \land \text{is-ct-entry-decl} \circ \text{s-ct-at}(\text{id(ctE)}(\text{ctDN})) \land \text{is-Q} \circ \text{s-ct-s}(\text{id(ctE)}(\text{ctDN})) \lor \text{is-Q} \circ \text{s-ct-body} \circ \text{id}(t) \land \text{is-Q} \circ \text{s-ct-at}(\text{id(ctE)}(\text{ctDN}))) \rightarrow \text{ct-null;}
\]
\[
\{ \text{update-ct-id}(\text{id}, \text{ct-af}(\text{*}, \text{id})) | \text{id} \in \text{ct-ident}(t) \land \text{is-Q} \circ \text{id}(\text{ctE}) \} \rightarrow T
\]
\[\text{T} \rightarrow \text{error} \]
for : is-ct-decl-part(t)
Ref. : ct-af 2-1(l)
Note: The condition represents the multiple declaration check, necessary when a declaration-part of an external program is to be interpreted. The first argument position of the address generating function ct-af represents scope information. The "*" indicates "global scope".

(15) \( \text{ct-ident}(t) = \{ \text{id} | \text{is-id}(\text{id}) \land \neg \text{is-Q} \circ \text{id}(t) \}\)
for : is-ct-decl-part(t)

(16) \( \text{update-ct-id}(\text{id}, \text{ad}) = \)
\[
\text{s-ct-e}; \mu(\text{ctE}; <\text{id}; \text{ad}>)
\]

(17) \( \text{ct-null =} \)
\[\text{PASS: Q} \]

(18) \( \text{int-ct-decl-part}(t) = \)
\[
\text{ct-null;}
\]
\[
\{ \text{int-ct-decl}(\text{id(ctE)}, \text{id}(t)) | \text{id} \in \text{ct-ident}(t) \}
\]
for: is-ct-decl-part(t)

(19) \( \text{int-ct-decl}(\text{ad}, \text{at}) = \)
\[
\text{is-LAB(at)} \rightarrow \text{s-ct-dn}; \mu(\text{ctDN}; <\text{s-ct-at}; \text{ad}; \text{LAB}>)
\]
\[
\text{is-ct-prop-var(at)} \rightarrow \text{update-ctat}(\text{ad}, \text{at})
\]
\[
\neg \text{is-Q} \circ \text{s-ct-entry-decl(at)} \land \neg \text{is-Q} \circ \text{s-ct-body(at)} \rightarrow \)
\[
\text{update-ctat}(\text{ad}, \text{s-ct-entry-decl(at)}); \text{update-ctp}(\text{ad}, \text{s-ct-body(at)})
\]
\[\text{cont'd} \]

8.3.1
\[ \text{update-ctat}(\text{ad}, \text{s-ct-entry-decl}(\text{at})) \]
\[ \text{update-ctp}(\text{ad}, \text{s-ct-body}(\text{at})) \]

for: \text{is-ct-decl}(\text{at})

\[ \text{update-ctat}(\text{ad}, \text{at}) = \]
\[ \text{s-ct-dn}: \mu(\text{ctDN}; \text{s-ct-at}; \text{ad}; \text{at}), \]
\[ \text{s-ct-act}; \text{ad}; \text{F}, \]
\[ \text{s-ct-cl}; \text{ad}; \text{F}) \]

for: \text{is-ct-prop-var}(\text{at}) \lor \text{is-ct-entry-decl}(\text{at})

\[ \text{update-ctp}(\text{ad}, \text{b}) = \]
\[ \text{is-id}(\text{b}) \rightarrow \text{s-ct-dn}: \mu(\text{ctDN}; \text{s-ct-s}; \text{ad}; \text{ct-af}(*, \text{b})), \]
\[ \text{T} \rightarrow \text{s-ct-dn}: \mu(\text{ctDN}; \text{s-ct-s}; \text{ad}; \text{ad}), \]
\[ \text{s-ct-cl}: \mu(\text{ctP}; \text{s-ct-body}; \text{ad}; \text{b}), \]
\[ \text{s-ct-dn}; \mu(\text{ctDN}; \text{s-ct-s}; \text{ad}; \text{ad}) \]

for: \text{is-ct-body}(\text{b}) \lor \text{is-id}(\text{b})

Ref.: \text{ct-af 2-1(1)}

Note: The proposition turns out to be true when in the \text{s-ct-body} component no procedure body was specified but rather the identifier which is associated in the declaration-part with the proper body. This occurs when a body has more than one entry name.

8.3.2 Interpretation of a (procedure-) text-part-list

The following instructions will execute the list of (procedure-) text-parts found in the text storage \text{ctT}. The execution is performed by the alternative interpretation of \text{ct}-statements and of the \text{ct}-text interspersed between the \text{ct}-statements, in the case of a text-part-list. Within a procedure body (\text{ctD} is not \text{\&}) no \text{ct}-text exists and \text{int-ct-text-part} replaces itself by a null-instruction.

After the whole (procedure-) text-part-list has been interpreted, the control dump is popped up, to continue the interpretation of the saved higher level (procedure-) text-part-list. If no \text{ctCD} exists and the interpretation took place outside procedures (\text{ctD} is \text{\&}), the computation is successfully terminated.
(22) \[ \text{int-ct-next-text-part} = \]
\[
\text{ctSC} \leftarrow \text{length}(\text{ctT}) \quad \text{s-ct-sc:ctSC+1} \]
\[
\text{s-ct-c: int-ct-next-text-part; int-ct-text-part;} \]
\[
\text{int-ct-prop-st(s-ct-prop-st,} \]
\[
\text{s-ct-st(elem(\text{ctSC+1,ctT}))} \]
\[
\neg \text{is-Q(\text{ctCD})} \quad \rightarrow \quad \text{s-ct-t: s-ct-t(\text{ctCD})} \]
\[
\text{s-ct-sc: s-ct-sc(\text{ctCD})} \]
\[
\text{s-ct-cd: s-ct-cd(\text{ctCD})} \]
\[
\text{s-ct-c: s-ct-c(\text{ctCD})} \]
\[
\text{is-Q(\text{ctD})} \quad \rightarrow \quad \text{s-ct-c: Q} \]
\[
\text{T} \quad \rightarrow \quad \text{error} \]

(23) \[ \text{int-ct-text-part} = \]
\[
\neg \text{is-Q(\text{ctD})} \quad \rightarrow \quad \text{ct-null} \]
\[
\text{T} \quad \rightarrow \quad \text{int-ct-text(<BLANK>, s-ct-text(elem(\text{ctSC,ctT})),*)} \]

Ref.: \text{int-ct-text 8-31(108)}

Note: The second argument of \text{int-ct-text}, namely "*", defines that the resulting output text must be transferred into \text{ctR} in distinction from the action required during the interpretation of function arguments (cf. 8-37(122))

(24) \[ \text{int-ct-prop-st(t) =} \]
\[
\text{is-ct-p-group(t) v is-ct-group(t) → int-ct-group(t)} \]
\[
\text{is-ct-text-part-list(t) v is-ct-p-text-part-list(t)} \quad → \quad \text{int-ct-group-text(t)} \]
\[
\text{is-ct-assign-st(t)} \quad → \quad \text{int-ct-assign-st(t)} \]
\[
\text{is-ct-goto-st(t)} \quad → \quad \text{int-ct-goto-st(t)} \]
\[
\text{is-ct-p-if-st(t) v is-ct-if-st(t)} \quad → \quad \text{int-ct-if-st(t)} \]
\[
\text{is-ct-null-st(t)} \quad → \quad \text{ct-null} \]
\[
\text{is-ct-return-st(t)} \quad → \quad \text{int-ct-return-st(t)} \]
\[
\text{is-ct-declare-st(t)} \quad → \quad \text{int-ct-decl-st(t)} \]
\[
\text{is-ct-act-st(t)} \quad → \quad \text{int-ct-act-st(t)} \]
\[
\text{is-ct-deact-st(t)} \quad → \quad \text{int-ct-deact-st(t)} \]
\[
\text{is-ct-include-st(t)} \quad → \quad \text{int-ct-include-st(t)} \]

cont'd
8.3.3 Interpretation of the group

The ct-group represents a concrete ct-do-group with iteration.

Metavariables

var is-ct-simple-var  
by is-ct-const ∨ is-Ω  
to is-ct-const ∨ is-Ω

(25) \text{int-ct-group}(t) =
\begin{align*}
\text{is-Ω}(a_t) \land & \text{is-ct-prop-var}s\text{-ct-at}\text{*}a_t(\text{ctDN}) \land s\text{-ct-decl}\text{*}a_t(\text{ctDN}) → \\
\text{int-ct-do}(s\text{-ct-var}s\text{-ct-iteration}(t),&\text{by},\text{to}); \\
\text{by:}\text{int-ct-byto-expr}(s\text{-ct-by-expr}s\text{-ct-iteration}(t)), \\
\text{to:}\text{int-ct-byto-expr}(s\text{-ct-to-expr}s\text{-ct-iteration}(t)), \\
\text{ct-assign}(s\text{-ct-var}s\text{-ct-iteration}(t),&v); \\
v:\text{ct-convert}(s\text{-ct-at}\text{*}a_t(\text{ctDN}),&r); \\
r:\text{int-ct-expr}(s\text{-ct-init-expr}s\text{-ct-iteration}(t)) \\
\end{align*}

T → \text{error}

where: a_t = s\text{-ct-id}s\text{-ct-var}s\text{-ct-iteration}(t)(\text{ctP})

for : is-ct-p-group(t) ∨ is-ct-group(t)

Ref.: \text{int-ct-expr} 8-10(34), ct-assign 8-10(33), ct-convert 8-15(55)

(26) \text{int-ct-byto-expr}(t) =
\begin{align*}
\text{is-Ω}(t) → \text{PASS}:\Omega \\
T → \text{int-ct-expr}(t) \\
\end{align*}

for : is-ct-expr(t) ∨ is-Ω(t)

Ref.: \text{int-ct-expr} 8-10(34)
(27) int-ct-do(var,by,to) =
    is-Ω(by) & is-Ω(to) →
    int-ct-group-text(s-ct-text-part-list,s-ct-prop-sts,
                       s-ct-st(elem(ctSC,ctT)))
    ~is-Ω(by) →
    iterate-ct-do(var,by,to,test);
    test:int-ct-expr(ct-cond-expr(var,by,to))

    T → error

Rёf.: int-ct-expr 8-10(34)

(28) ct-cond-expr(var,by,to) =
    is-Ω(to) → \( \mu_o(\langle s-ct-da:BIT, s-ct-vr:ct-represent(BIT,1-CHAR) \rangle) \),
    T → \( \mu_o(\langle s-ct-op:OR, \langle s-ct-opd-1: \mu_o(\langle s-ct-op:AND, \langle s-ct-opd-1:by, \langle s-ct-opd-2:ct-zero, \langle s-ct-opd-2:to \rangle \rangle \rangle \rangle ) \rangle \),
    \( \mu_o(\langle s-ct-opd-2: \mu_o(\langle s-ct-op:AND, \langle s-ct-opd-1:by, \langle s-ct-opd-2:ct-zero, \langle s-ct-opd-2:to \rangle \rangle \rangle ) \rangle \),
    \( \mu_o(\langle s-ct-opd-2: \mu_o(\langle s-ct-op:AND, \langle s-ct-opd-1:by, \langle s-ct-opd-2:ct-zero, \langle s-ct-opd-2:to \rangle \rangle \rangle ) \rangle \),
    \( \mu_o(\langle s-ct-opd-2: \mu_o(\langle s-ct-op:AND, \langle s-ct-opd-1:by, \langle s-ct-opd-2:ct-zero, \langle s-ct-opd-2:to \rangle \rangle ) \rangle \),
    \( \mu_o(\langle s-ct-opd-2: \mu_o(\langle s-ct-op:AND, \langle s-ct-opd-1:by, \langle s-ct-opd-2:ct-zero, \langle s-ct-opd-2:to \rangle \rangle ) \rangle \),
    \( \mu_o(\langle s-ct-opd-2: \mu_o(\langle s-ct-op:AND, \langle s-ct-opd-1:by, \langle s-ct-opd-2:ct-zero, \langle s-ct-opd-2:to \rangle \rangle ) \rangle \),

Rёf.: ct-represent 2-2(5)

(29) ct-zero =
    \( \mu_o(\langle s-ct-da:INTG, s-ct-vr:ct-represent(INTG,0) \rangle) \)

Rёf.: ct-represent 2-2(5)
8.3.4 Assignment statement, evaluation of expressions, and conversions.

Metavariables

\begin{align*}
\text{opd1} & \quad \text{is-ct-const} \\
\text{opd2} & \quad \text{is-ct-const} \\
\text{opd} & \quad \text{is-ct-const} \\
\text{op} & \quad \text{is-ct-infix-op} \lor \text{is-ct-prefix-op} \\
\end{align*}

\begin{align*}
(32) \quad \text{int-ct-assign-st}(t) =
& -\text{is-Q}(a_t) \land \text{is-ct-prop-vars}s-ct-at\!a_t(ctDN) \land s-ct-dcl\!a_t(ctDN) \quad \text{ct-assign}(s-ct-lp(t),v) \\
& v: \text{ct-convert}(s-ct-at\!a_t(ctDN),r) \\
& r: \text{int-ct-expr}(s-ct-rp(t)) \\
& T \quad \text{error}
\end{align*}

\text{cont'd}
where: \( a_t = s-ct-id \cdot s-ct-lp(t)(ctE) \)
for: is-ct-assign-st(t)

(33) \( \text{ct-assign}(t,v) = s-ct-dn: \mu(ctDN; s-ct-s\cdot s-ct-id(t)(ctE)):\!\!s-ct-vr(v)) \)
for: is-ct-simple-var(t), is-ct-const(v)

(34) \( \text{int-ct-expr}(t) = \)
\( \begin{align*}
\text{is-ct-infix-expr}(t) & \Rightarrow \text{int-ct-infix-expr}(\text{opd-1}, \text{opd-2}, s-ct-op(t)); \\
\text{opd-1}: \text{int-ct-expr}(s-ct-opd-1(t)), \\
\text{opd-2}: \text{int-ct-expr}(s-ct-opd-2(t)) \\
\text{is-ct-prefix-expr}(t) & \Rightarrow \text{int-ct-prefix-expr}(\text{opd}, s-ct-op(t)); \\
\text{opd}: \text{int-ct-expr}(s-ct-opd(t)) \\
\text{is-ct-paren-expr}(t) & \Rightarrow \text{int-ct-expr}(s-ct-opd(t)) \\
\text{is-ct-ref}(t) & \Rightarrow \text{int-ct-ref}(t) \\
\text{is-ct-const}(t) & \Rightarrow \text{PASS:ct-const-test}(t)
\end{align*} \)
for: is-ct-expr(t)
Ref.: int-ct-ref 8-18(66)

(35) \( \text{ct-const-test}(t) = \)
\( \begin{align*}
\text{is-INTG}\cdot s-ct-da(t) & \Rightarrow \\
\mu(t; s-ct-vr: \text{ct-represent}(\text{INTG}, \text{ct-value}(\text{INTG}, s-ct-vr(t)))) \\
T & \Rightarrow \\
\mu(t; s-ct-vr: \text{ct-representlist}(s-ct-da(t), \text{ct-valuelist}(s-ct-da(t), s-ct-vr(t))))
\end{align*} \)
for: is-ct-const(t)
Ref.: ct-represent 2-2(5) ct-value 2-2(6)
Note: This function checks whether the both components of a constant, i.e. the data attribute and the value representation, match. For the case that the constant represents an integer greater or equal 10^n, the function ct-value is undefined (error).
(36) \( ct\text{-}representlist(at, v) = \)
\[
\begin{align*}
\text{is-}<>(v) & \rightarrow <> \\
T & \rightarrow \langle ct\text{-}represent(at, head(v)) \rangle \cup ct\text{-}representlist(at, \text{tail}(v))
\end{align*}
\]
Ref.: \( ct\text{-}represent \ 2\-2(5) \)

(37) \( ct\text{-}valuelist(at, vr) = \)
\[
\begin{align*}
\text{is-}<>(vr) & \rightarrow <> \\
T & \rightarrow \langle ct\text{-}value(at, head(vr)) \rangle \cup ct\text{-}valuelist(at, \text{tail}(vr))
\end{align*}
\]
Ref.: \( ct\text{-}value \ 2\-2(6) \)

(38) \( \text{int-ct-infix-expr}(opd_1, opd_2, op) = \)
\[
\begin{align*}
\text{ct-infix-int}(op_1, op_2, op); \\
op_1 & : ct\text{-}convert(ct\text{-}target(s\text{-}ct\text{-}da(op_1), s\text{-}ct\text{-}da(opd_2), op), op_1), \\
op_2 & : ct\text{-}convert(ct\text{-}target(s\text{-}ct\text{-}da(op_1), s\text{-}ct\text{-}da(opd_2), op), op_2)
\end{align*}
\]
for : \( is\text{-}ct\text{-}infix-op(op) \)

(39) \( ct\text{-}target(da_1, da_2, op) = \)
\[
\begin{align*}
is\text{-}ct\text{-}arith\text{-}op(op) \lor \text{op} \in \{\text{PLUS, MINUS}\} & \rightarrow \text{INTG} \\
is\text{-}ct\text{-}comp\text{-}op(op) \land (is\text{-}INTG(da_1) \lor \text{is\text{-}INTG(da_2)}) & \rightarrow \text{INTG} \\
is\text{-}ct\text{-}comp\text{-}op(op) \land \text{is\text{-}BIT(da_1) \& is\text{-}BIT(da_2)} & \rightarrow \text{BIT} \\
is\text{-}ct\text{-}comp\text{-}op(op) & \rightarrow \text{CHAR} \\
is\text{-}ct\text{-}bit\text{-}op(op) \lor \text{is\text{-}NOT(op)} & \rightarrow \text{BIT} \\
is\text{-}CAT(op) \& \text{is\text{-}BIT(da_1) \& is\text{-}BIT(da_2)} & \rightarrow \text{BIT} \\
is\text{-}CAT(op) & \rightarrow \text{CHAR}
\end{align*}
\]
for : \( is\text{-}INTG(da_i) \lor \text{is\text{-}BIT(da_i) \lor is\text{-}CHAR(da_i) \lor is\text{-}*(da_i), \ i=1,2} \)

(40) \( ct\text{-}infix\text{-}int}(opd_1, opd_2, op) = \)
\[
\begin{align*}
is\text{-}CAT(op) & \rightarrow \text{ct\text{-}pass\text{-}res}(s\text{-}ct\text{-}da(opd_1), \\
& \quad s\text{-}ct\text{-}vr(opd_1) \cap s\text{-}ct\text{-}vr(opd_2)) \\
is\text{-}ct\text{-}comp\text{-}op(op) & \rightarrow \text{ct\text{-}pass\text{-}res}(\text{BIT}, \langle \text{ct\text{-}represent} \\
& \quad (\text{BIT}, \text{ct\text{-}comp\text{-}res}(opd_1, opd_2, op)) \rangle) \\
is\text{-}ct\text{-}bit\text{-}op(op) & \rightarrow \text{ct\text{-}pass\text{-}res}(\text{BIT}, \text{ct\text{-}representlist} \\
& \quad (\text{BIT}, \text{ct\text{-}bit\text{-}res}(s\text{-}ct\text{-}vr(opd_1), \\
& \quad s\text{-}ct\text{-}vr(opd_2), op))) \\
\end{align*}
\]
cont'd
is-ct-arithmetic-op(op) → ct-pass-res(INTG, ct-represent(INTG, ct-arithmetic-res (ct-value(INTG, s-ct-vr(opd1)), ct-value(INTG, s-ct-vr(opd2)), op)))

for: is-ct-infix-op(op)
Ref.: ct-represent 2-2(5), ct-value 2-2(6)

(41) ct-pass-res(da, vr) =
      PASS: μ₀(〈s-ct-da:da〉, 〈s-ct-vr:vr〉)

(42) ct-comp-res(opd1, opd2, op) =
      is-ct-string(opd1) →
      ct-string-comp(s-ct-vr(opd1), s-ct-vr(opd2), op, s-ct-da(opd1))
      T →
      ct-arithmetic-comp(ct-value(INTG, s-ct-vr(opd1)), ct-value(INTG, s-ct-vr(opd2)), op)

for: is-ct-comp-op(op)
Ref.: ct-value 2-2(6)

(43) ct-string-comp(vr1, vr2, op, da) =
      is-<>(vr1 vr2) → (op ∈ {EQ, GE, LE} → 1-CHAR,
      T → O-CHAR)
      ct-string-val(da, hd₁) ≠ ct-string-val(da, hd₂) →
      ct-arithmetic-comp(ct-string-val(da, hd₁), ct-string-val(da, hd₂), op)
      is-<>(vr1) → ct-string-comp(<> , tail(vr2), op, da)
      is-<>(vr2) → ct-string-comp(tail(vr1), <>, op, da)
      T → ct-string-comp(tail(vr1), tail(vr2), op, da)

where: hd₁ = (is-<>(vr1) → ₀, T → head(vr1))
       hd₂ = (is-<>(vr2) → ₀, T → head(vr2))

for: is-ct-vr-list(vri), i=1,2
      is-ct-comp-op(op), is-BIT(da) ∨ is-CHAR(da)

(44) ct-string-val(da, v) =
      -is-₀(v) → (is-BIT(da) → digit-val ct-value(BIT, v),
                 is-CHAR(da) → ct-collect ct-value(CHAR, v))
      is-BIT(da) → ₀
      is-CHAR(da) → ct-collect(BLANK)

cont'd
for : is-BIT(da) v is-CHAR(da), is-ct-vr(v) v is-O(v)
Ref.: ct-collat 2-1(2), digit-val 5-13(46), ct-value 2-2(6)

(45) ct-arith-comp(v1,v2,op) =
    (op ∈ {EQ,GE,LE} & v1=v2) v (is-NE(op) & v1≠v2) v
    (op ∈ {GT,GE} & v1>v2) v (op ∈ {LT,LE} & v1<v2) ---→ 1-CHAR

T ---→ 0-CHAR

for: is-intg-val(vi), i=1,2
    is-ct-comp-op(op)

(46) ct-bit-res(vr1,vr2,op) =
    is<>(vr1 ⊗ vr2) ---<>
    is<>(vr2) ---→ ct-bit-res(vr2,vr1,op)
    is<>(vr1) ---→
    <ct-log-string(0-CHAR,ct-value(BIT,head(vr2)),op)> ⊗
    ct-bit-res(<>,tail(vr2),op)

T ---→
    <ct-log-string(ct-value(BIT,head(vr1)),ct-value(BIT,head(vr2)),op)> ⊗
    ct-bit-res(tail(vr1),tail(vr2),op)

for : is-ct-vr-list(vri), i=1,2
    is-ct-bit-op(op)
Ref.: ct-value 2-2(6)

(47) ct-log-string(v1,v2,op) =
    (is-AND(op) & is-1-CHAR(v1) & is-1-CHAR(v2)) v
    (is-OR(op) & (is-1-CHAR(v1) v is-1-CHAR(v2))) ---→ 1-CHAR

T ---→ 0-CHAR

for: (is-1-CHAR v is-O-CHAR)(vi), i=1,2
    is-ct-bit-op(op)

(48) ct-arith-res(v1,v2,op) =
    abs(ct-arith-val(v1,v2,op)) > 10 ↑ n ---→
    ct-truncate(ct-arith-val(v1,v2,op),n)

T ---→ ct-arith-val(v2,v2,op)

for : is-intg-val(vi), i=1,2
    is-arith-op(op)
Ref.: n 2-2(4), ct-truncate 2-2(3)
(49) ct-arith-val(v1,v2,op) =
  is-ADD(op) → v1+v2
  is-SUBTR(op) → v1 - v2
  is-MULT(op) → v1.v2
  is-DIV(op) → ct-div(v1,v2)

(50) ct-div(v1,v2) =
  sign(v1) = sign(v2) → ct-abs-div(abs(v1),abs(v2))
  T → -ct-abs-div(abs(v1),abs(v2))

(51) ct-abs-div(v1,v2) =
  v1<v2 → 0
  T → ct-abs-div(v1 - v2,v2)+1

(52) int-ct-prefix-expr(opd,op) =
  ct-prefix-int(opd,op);
  opd:ct-convert(ct-target(*,*,op),opd)
  for: is-ct-prefix-op(op)

(53) ct-prefix-int(opd,op) =
  is-PLUS(op) → PASS:opd
  is-MINUS(op) → ct-pass-res(INTG,ct-represent(INTG,-ct-value(INTG,
                           s-ct-vr(opd))))
  is-NOT(op) → ct-pass-res(BIT,ct-representlist(BIT,ct-not-val*
                           s-ct-vr(opd)))
  for: is-ct-prefix-op(op)
  Ref.: ct-represent 2-2(5), ct-value 2-2(6)

(54) ct-not-val(vr) =
  is-<>(vr) → <>
  is-1-CHAR ct-value(BIT,head(vr)) → <O-CHAR> ^ct-not-val·tail(vr)
  T → <l-CHAR> ^ct-not-val·tail(vr)
  Ref.: ct-value 2-2(6)
(55) \[ \text{ct-convert}(da, opd) = \]
\[ s-ct-da(opd) = da \rightarrow \text{PASS:opd} \]
\[ \text{is-INTG}(da) \land \text{is-CHAR}s-ct-da(opd) \rightarrow \]
\[ \text{ct-pass-res}(\text{INTG}, \text{ct-represent}(\text{INTG}, \text{ct-char-to-int}
\text{(ct-valuelist(CHAR,s-ct-vr(opd))))}) \]
\[ \text{is-INTG}(da) \land \text{is-BIT}s-ct-da(opd) \rightarrow \]
\[ \text{ct-pass-res}(\text{INTG}, \text{ct-represent}(\text{INTG}, \text{ct-bit-to-int}
\text{(ct-valuelist(BIT,s-ct-vr(opd))))}) \]
\[ \text{is-CHAR}(da) \land \text{is-INTG}s-ct-da(opd) \rightarrow \]
\[ \text{ct-pass-res}(\text{CHAR}, \text{ct-representlist}(\text{CHAR}, \text{ct-int-to-char}
\text{(ct-value(\text{INTG}, s-ct-vr(opd))))}) \]
\[ \text{is-BIT}(da) \land \text{is-INTG}s-ct-da(opd) \rightarrow \]
\[ \text{ct-pass-res}(\text{BIT}, \text{ct-representlist}(\text{BIT}, \text{ct-int-to-string}
\text{(abs(ct-value(\text{INTG}, s-ct-vr(opd))),2)})) \]
\[ \text{is-CHAR}(da) \land \text{is-BIT}s-ct-da(opd) \rightarrow \]
\[ \text{ct-pass-res}(\text{CHAR}, \text{ct-representlist}(\text{CHAR}, \text{ct-valuelist}(\text{BIT},s-ct-vr(opd)))) \]
\[ \text{is-BIT}(da) \land \text{is-CHAR}s-ct-da(opd) \rightarrow \]
\[ \text{ct-pass-res}(\text{BIT}, \text{ct-char-to-bit}s-ct-vr(opd)) \]
\[ \text{for: is-INTG}(da) \lor \text{is-CHAR}(da) \lor \text{is-BIT}(da) \]
Ref.: \text{ct-represent}\ 2-2(5), \text{ct-value}\ 2-2(6) 

(56) \[ \text{ct-char-to-int}(v) = \]
\[ \text{is-BLANK}\cdot \text{last-el}(v) \rightarrow \text{ct-char-to-int}\cdot \text{headlist}(v) \]
\[ \text{is-digit}\cdot \text{last-el}(v) \rightarrow \]
\[ \text{(is-PLUS}\cdot \text{ct-sign}(v) \rightarrow (\text{ct-int-val}(v) \geq 10 \uparrow n \rightarrow \]
\text{ct-truncate(\text{ct-int-val}(v),n),}
\[ \text{T} \rightarrow \text{ct-int-val}(v)), \]
\[ \text{is-MINUS}\cdot \text{ct-sign}(v) \rightarrow (\text{ct-int-val}(v) \geq 10 \uparrow n \rightarrow \]
\[ - \text{ct-truncate(\text{ct-int-val}(v),n),}
\[ \text{T} \rightarrow - \text{ct-int-val}(v)), \]
\[ \text{T} \rightarrow \text{error} \]
\[ \text{T} \rightarrow \text{error} \]
\[ \text{for: is-char-val-list}(v) \]
Ref.: \text{headlist}\ 2-4(10), \text{last-el}\ 2-4(11), \text{n}\ 2-2(4), \text{ct-truncate}\ 2-2(3) 

8.3.4
(57) \( ct\text{-}sign(v) = \)
\[
\text{is-BLANK-list}(v) \rightarrow \text{PLUS}
\]
\[
\text{is-dig}it\text{-}last\text{-}el(v) \rightarrow \text{ct\text{-}sign\text{-}headlist}(v)
\]
\[
\text{last\text{-}el}(v) \in \{\text{PLUS}, \text{MINUS}\} \& \text{is-BLANK-list\text{-}headlist}(v) \rightarrow \text{last\text{-}el}(v)
\]
\[
T \rightarrow \emptyset
\]

for: is-char-val-list(v)
Ref.: last-el 2-4(11), headlist 2-4(10)

(58) \( ct\text{-}int\text{-}val(v) = \)
\[
\text{is-dig}it\text{-}last\text{-}el(v) \rightarrow \text{ct\text{-}int\text{-}val\text{-}headlist}(v) \cdot 10 + \text{digit\text{-}val\text{-}last\text{-}el}(v)
\]
\[
T \rightarrow 0
\]

Ref.: digit-val 5-13(46), last-el 2-4(11), headlist 2-4(10)

(59) \( ct\text{-}bit\text{-}to\text{-}int(v) = \)
\[
\text{ct\text{-}bit\text{-}val}(v) \geq 10 \uparrow n \rightarrow \text{ct\text{-}truncate}(\text{ct\text{-}bit\text{-}val}(v), n)
\]
\[
T \rightarrow \text{ct\text{-}bit\text{-}val}(v)
\]

Ref.: n 2-2(4), ct-truncate 2-2(3)

(60) \( ct\text{-}bit\text{-}val(v) = \)
\[
\text{is-<>}(v) \rightarrow 0
\]
\[
T \rightarrow \text{ct\text{-}bit\text{-}val\text{-}headlist}(v) \cdot 2 + \text{digit\text{-}val\text{-}last\text{-}el}(v)
\]

Ref.: digit-val 5-13(46), headlist 2-4(10), last-el 2-4(11)

(61) \( ct\text{-}int\text{-}to\text{-}char(v) = \)
\[
v \geq 0 \rightarrow \text{ct\text{-}blank\text{-}fill}(\text{ct\text{-}int\text{-}to\text{-}string}(v, 10))
\]
\[
T \rightarrow \text{ct\text{-}blank\text{-}fill}(<\text{MINUS}> \cap \text{ct\text{-}int\text{-}to\text{-}string}(-v, 10))
\]

for: is-intg-val(v)

(62) \( ct\text{-}int\text{-}to\text{-}string(v,b) = \)
\[
v < b \rightarrow <\text{char\text{-}val}(v)>
\]
\[
T \rightarrow \text{ct\text{-}int\text{-}to\text{-}string}(\text{ct\text{-}abs\text{-}div}(v,b),b) \cap \text{<char\text{-}val}(v - b \cdot \text{ct\text{-}abs\text{-}div}(v,b))>
\]
(63) char-val(v) =

\[
\begin{align*}
    v=0 & \rightarrow 0-\text{CHAR} \\
    v=1 & \rightarrow 1-\text{CHAR} \\
    v=2 & \rightarrow 2-\text{CHAR} \\
    v=3 & \rightarrow 3-\text{CHAR} \\
    v=4 & \rightarrow 4-\text{CHAR} \\
    v=5 & \rightarrow 5-\text{CHAR} \\
    v=6 & \rightarrow 6-\text{CHAR} \\
    v=7 & \rightarrow 7-\text{CHAR} \\
    v=8 & \rightarrow 8-\text{CHAR} \\
    v=9 & \rightarrow 9-\text{CHAR}
\end{align*}
\]

(64) ct-blank-fill(v) =

\[
\begin{align*}
    \text{length}(v) = n+3 & \rightarrow v \\
    \text{length}(v) < n+3 & \rightarrow \text{ct-blank-fill}(\langle \text{BLANK} \rangle \, v) \\
    T & \rightarrow \text{error}
\end{align*}
\]

(65) ct-char-to-bit(vr) =

\[
\begin{align*}
    \text{is-=} (vr) & \rightarrow \langle \rangle \\
    \text{ct-value}(\text{CHAR}, \text{head}(vr)) \in \{0-\text{CHAR}, 1-\text{CHAR}\} & \rightarrow \langle \text{ct-represent}(\text{BIT}, \text{ct-value}(\text{CHAR}, \text{head}(vr))) \rangle \cap \text{ct-char-to-bit}(\text{tail}(vr)) \\
    T & \rightarrow \text{error}
\end{align*}
\]

Ref.: ct-value 2-2(6), ct-represent 2-2(5)
8.3.5 Evaluation of references appearing in ct-expressions

There are three different types of ct-references to be handled:

Reference of the built-in function SUBSTR
Reference of a ct-variable
Reference of a ct-function procedure.

Metavariables

id is-id
ad is-ad
p is-ct-descr

(66) \text{int-ct-ref}(t) =

\[(\text{is-}Q(\text{gt}_t) \land \text{is-}Q(a_t) \land \text{id}_t = \text{mk-id}(\text{SUBSTR})) \lor \\
(\neg \text{is-}Q(\text{gt}_t) \land \text{is-BUILTIN}\text{-ct-at}\text{gt}_t(\text{ctun}) \land \\
(\text{is-}Q(a_t) \lor \text{is-BUILTIN}\text{-ct-at}(\text{dnt})) \land 2 \leq \text{length}\text{-ct-arg-list}(t) \leq 3 \rightarrow \\
\text{int-ct-built-in(}_{\text{id}_t,\text{agl});} \\
\text{agl:\text{ct-dummy-assign-list(id}_t,\text{argl);} \\
\text{argl:\text{int-ct-arg-list(s-ct-arg-list(}_{t},\text{ct-built-in}_{par(}_{\text{id}_t},<>);} \\
\text{update-g-cte(id}_t,\text{ct-af(}_{*},\text{id}_t))} \\
\text{is-ct-simple-var(}_{t} \land \neg \text{is-}Q(a_t) \land \\
\text{is-ct-prop-vars\text{-ct-at(}_{dnt}) \land \text{is-ct-dcl(dnt)} \land \neg \text{is-}Q\text{-ct-s(dnt)} \land \text{PASS: } \mu_0 \langle <\text{ct-das-ct-at(}_{dnt}>), \\
<\text{ct-vr:ct-s(}_{dnt}> angle} \\
\neg \text{is-}Q(a_t) \land \text{is-ct-entry-decl\text{-ct-at(}_{dnt}) \land \text{is-ct-dcl(dnt)} \land \\
\neg \text{is-}Q\text{-ct-s(dnt)} \land \neg \text{is-}Q\text{-ct-s(dnt)}(\text{ctp}) \land \\
\neg \text{is-ct-cld(s-ct-s(}_{dnt})(\text{ctp}) \land \\
\text{length}\text{-ct-arg-list}(t) = \text{length}\text{-ct-par-list}\text{-ct-at(}_{dnt}) \rightarrow \\
\text{ct-function-val[a];} \\
\text{int-ct-call(}_{s-ct-at(}_{dnt},s-ct-s(}_{dnt},\text{agl},a);} \\
a:_\text{ct-un-name}, \\
\text{agl:\text{ct-dummy-assign-list(id}_t,\text{argl);} \\
\text{argl:\text{int-ct-arg-list(s-ct-arg-list(}_{t},s-ct-par-list\text{-ct-at(}_{dnt}),} \\
<>)} \\
\rightarrow \text{error} \]
\[ g_{a_t} = s\text{-ct-id}(t)(\text{get-g-cte}(\xi)(\eta)) \]
\[ d_{n_t} = s\text{-ct-id}(t)(\text{ctE})(\text{ctDN}) \]

for: is-ct-ref(t)

Ref.: ct-af 2-1(1) , mk-id 2-3(9) , int-ct-call 8-29(102)

Note: ct. 7.4.3.

(67) get-g-cte(st) =

\[ \neg \text{is-Q}\cdot s\text{-ct-d}(st) \rightarrow (\text{get-g-cte}s\text{-ct-d}(st))\cdot s\text{-ct-d} \]

T \rightarrow s\text{-ct-e} 

for: is-ct-state(st) v is-ct-d(st)

Note: This function composes a selector, which, if applied to the current state yields the outermost environment, i.e. the environment stored in the bottom of the dump.

(68) ct-built-in-par(id) =

id = mk-id(SUBSTR) \rightarrow \langle \text{CHAR}, \text{INTG}, \text{INTG} \rangle 

Ref.: mk-id 2-3(9)

(69) update-g-cte(id, ad) =

\[ \neg \text{is-Q}\cdot id(\text{get-g-cte}(\xi)(\eta)) \rightarrow \text{ct-null} \]

T \rightarrow \text{ctI}: \mu(\xi_1)\langle id\cdot(\text{get-g-cte}(\xi))\cdot ad\rangle,

\[ \langle s\text{-ct-at}\cdot ad\cdot s\text{-ct-dn: BUILTIN} \rangle, \]
\[ \langle s\text{-ct-act}\cdot ad\cdot s\text{-ct-dn:F} \rangle, \]
\[ \langle s\text{-ct-decl}\cdot ad\cdot s\text{-ct-dn:T} \rangle \]

for: id = mk-id(SUBSTR)

Note: By this instruction the identifier mk-id(SUBSTR) gets an entry in the outermost environment. Under the same address a proper entry is made in ctDN giving mk-id(SUBSTR) the denotation of the built-in function.

(70) int-ct-arg-list(agl,pal,r) =

is-\langle\rangle(agl) \rightarrow \text{PASS}: r 

T \rightarrow \text{int-ct-arg-list}(\text{tail}(agl),\text{tail}(pal),rl);

rl:ct-pass-cat(r,v);

v: \text{int-ct-arg}(\text{head}(agl),\text{head}(pal)) 

cont'd
for: is-ct-expr-list(aql), is-ct-descr-list(pal), is-list(r)

(71) int-ct-arg(t,p) =

\[\text{is-ct-simple-var}(t) \& \neg \text{is-\textbullet}(a_t) \& \text{is-ct-prop-var}s-ct-at\cdot a_t(tDN) \& s-ct-dcl\cdot a_t(tDN) \& (\text{ct-match}(s-ct-at\cdot a_t(tDN),p) \lor \text{is-\textbullet}(p)) \rightarrow \text{PASS}:a_t\]

\[\text{is-ct-expr}(t) \rightarrow \text{ct-convert}(ct-pat(v,p),v);\]

\[v:\text{ct-bit-test}(r,p);\]

\[r:\text{int-ct-expr}(t)\]

where: \[a_t = s-ct-id(t)(ctE)\]

for: is-ct-expr(t)

Ref.: ct-convert 8-15(55), int-ct-expr 8-10(34), ct-pat 8-37(123)

Note: The instruction passes either an address ("by name" case) or a ct-constant ("by value").

(72) ct-bit-test(r,p) =

\[\neg \text{is-BIT}s-ct-da(r) \lor \neg \text{is-\textbullet}(p) \rightarrow \text{PASS}:r\]

\[T \rightarrow \text{error}\]

for: is-ct-const(r)

Note: The case where the evaluated argument is a bit string and the corresponding parameter descriptor is "\textbullet", i.e. was not specified, must lead to an error.

(73) ct-pass-cat(r,t) =

\[\text{PASS}:r \leftarrow t\]

for: is-list(r)

(74) ct-match(at,p) =

\[\text{is-INTG}(at) \& \text{is-INTG}(p) \lor \text{is-CHAR}(at) \& \text{is-CHAR}(p)\]

for: is-ct-prop-var(at)
(75) \( \text{ct-dummy-assign-list}(\text{id}, \text{agl}) = \)

\[
\text{ct-pass}(\text{agl});
\{ \text{elem}(i) \cdot \text{agl}; \text{ct-dummy-assign}(\text{ct-af}(\text{id}, i), \text{elem}(i, \text{agl})) \mid
\text{ls-length}(\text{agl}) \land \neg \text{is-ad}(\text{elem}(i, \text{agl})) \}
\]

Ref.: ct-af 2-1(1)

Note: For those arguments where no simple address passing occurred, the creation of a dummy entry in \text{ctDN} will take place. The necessary address is generated by \text{ct-af}, mapping the entry name of the function together with the element number of the evaluated argument onto that address.

(76) \( \text{ct-pass}(t) = \)

\[
\text{PASS}:t
\]

(77) \( \text{ct-dummy-assign}(\text{ad}, \text{val}) = \)

\[
\text{PASS} : \text{ad}
\]

\[
\text{s-ct-dn} : (\text{ctDN} ; <s-\text{ct-at} : \text{ad} ; s-\text{ct-da}(\text{val}) > ,
<s-\text{ct-dcl} : \text{ad} ; T> ,
<s-\text{ct-s} : \text{ad} ; s-\text{ct-vr}(\text{val}) > )
\]

for: \( \text{is-ct-const}(\text{val}) \land \neg \text{is-BIT} \cdot s-\text{ct-da}(\text{val}) \)

Note: The activate status of the denotation is left 0, because it is of no use within a procedure body.

(78) \( \text{ct-un-name} = \)

\[
\text{PASS} : n_{\text{ctUN}}
\]

\[
\text{s-ct-un} : n_{\text{ctUN}} + 1
\]

Note: This instruction returns the \( n_{\text{ctUN}} \)-th unique name out of the infinite, ordered set of unique names, and increases the unique name counter by 1. Note, that the set of the unique names has no elements in common with the range of the function \text{ct-af}.
(79) \( \text{ct-function-val}(a) = \)

\[
\mu_o <s-ct-da:s-ct-at•a(ctDN)>,
<s-ct-vr:is-ct-s•a(ctDN)>\]

Note: This instruction is used to pass the value of a function procedure which was stored in the denotation directory at the time the return statement was interpreted.

(80) \( \text{int-ct-built-in}(id,agl) = \)

\[
is-<>\text{tail\_tail}(agl) \rightarrow \]

\[
\text{ct-pass-res}(\text{CHAR},\text{ct-substr-val}(s-ct-s(head(agl)(ctE)(ctDN)),
\text{ct-value}(\text{INT},s-ct-s(elem(2)(agl)(ctE)(ctDN))),*)
\]

\[
T \rightarrow \text{ct-pass-res}(\text{CHAR},\text{ct-substr-val}(s-ct-s(head(agl)(ctE)(ctDN))),
\text{ct-value}(\text{INT},s-ct-s(elem(2)(agl)(ctE)(ctDN))),
\text{ct-value}(\text{INT},s-ct-s(elem(3)(agl)(ctE)(ctDN))))
\]

for: \( \text{is-ad-list}(agl) \)

Ref.: \( \text{ct-pass-res} \ 8-12(41), \text{ct-value} \ 2-2(6) \)

(81) \( \text{ct-substr-val}(vr,i,j) = \)

\[
i=1 \rightarrow \text{ct-collect}(vr,j)
\]

\[
T \rightarrow \text{ct-substr-val}(\text{tail}(vr),i-1,j)
\]

for: \( \text{is-ct-vr-list}(vr), \text{is-intg-val}(i), \text{is-intg-val}(j) \vee \text{is-•}(j) \)

(82) \( \text{ct-collect}(vr,j) = \)

\[
is-•(j) \rightarrow vr
\]

\[
j=\emptyset \rightarrow <>
\]

\[
T \rightarrow <\text{head}(vr)> \text{ct-collect}(\text{tail}(vr),j-1)
\]

for: \( \text{is-ct-vr-list}(vr), \text{is-intg-val}(j) \vee \text{is-•}(j) \)
8.3.6 Goto statement

Metavariable:

\( cl \) is-integer-list

\( (83) \) \( \text{int-ct-goto-st}(t) = \)

\( \neg \text{is-}O(s-s-\text{ct-label}(t)(cT)) \) \& \( \text{is-LAB\^}s-s-\text{ct-at}(s-s-\text{ct-label}(t)(cT)(cD\text{N})) \) \rightarrow

\( \text{int-ct-goto}(t,cT,\text{ct-label}(s-s-\text{ct-label}(t),cT,1)), \)

T \rightarrow \text{error}

for: \( \text{is-ct-goto-st}(t) \)

\( (84) \) \( \text{ct-label}(id,t,sc) = \)

\( \neg \text{is-}O(s-s-\text{ct-label}(t)(cT)) \) \& \( \text{is-LAB\^}s-s-\text{ct-at}(cT) \) \&

\( \neg \text{is-}O(s-s-\text{ct-label}(id,pst_t),1) \) \&

\( \neg \text{is-}O(s-s-\text{ct-if-st}(pst_t) \lor s-s-\text{ct-p-if-st}(pst_t)) \) \&

\( \neg \text{is-}O(s-s-\text{ct-then}(pst_t),1) \) \&

\( \neg \text{is-}O(s-s-\text{ct-else}(pst_t),1) \)

\( \neg \text{is-}O(s-s-\text{ct-prop-st}(pst_t)) \)

where: \( \text{pst}_t = s-s-\text{prop-st}(s-s-\text{ct-st}(elem(sc,t))) \)

for: \( \text{is-id}(id), \text{is-ct-t}(t), \text{is-intg-val}(sc) \)

Note: The function yields a list of statement counters giving the position
where the label is to be found in the (procedure-) text-part-list stor-
ed in \( \text{ctT} \). The function will return the empty list "<>" if the label
was not found.
(85) \[ \text{int-ct-goto}(t, cl) = \]
\begin{align*}
\text{is-<>}(cl) & \rightarrow \text{s-ct-sc}\text{-head}(cl) \\
\text{is-Ω}(ctCD) & \rightarrow \text{s-ct-t: } \text{s-ct-t}(ctCD) \\
\end{align*}
\begin{align*}
\text{s-ct-cd: } & \text{s-ct-cd}(ctCD) \\
\text{s-ct-c: } & \text{int-ct-goto-st}(t) \\
T & \rightarrow \text{error} \\
\end{align*}

for: \text{is-ct-goto-st}(t)

Note: If no label was found by \text{ct-labelelf}, i.e. \text{is-<>}(cl), and \text{ctCD} exists, the text storage component of \text{ctCD} is installed in \text{ctT}, the control dump is popped up, and the goto statement is executed again. If \text{ctCD} is \text{Ω}, the goto statement is in error. It indicates that the outermost text-part-list of the program itself or the outermost procedure-text-part-list of a body was scanned and the label still was not found. (Note that when entering a procedure body the new \text{ctCD} is set to \text{Ω}).

(86) \[ \text{ct-elem-execute}(cl) = \]
\begin{align*}
\text{int-ct-next-text-part; } \\
\text{int-ct-text-part; } \\
\text{ct-if-st}(pst_t, \text{tail}(cl)) \\
T & \rightarrow \text{int-ct-next-text-part; } \\
\text{int-ct-text-part; } \\
\text{ct-update-cd}(pst_t, \text{tail}(cl)) \\
\end{align*}

where: \(pst_t = \text{s-ct-prop-st: s-ct-st}(\text{elem(head(cl), ctT)})\)

Ref.: \text{int-ct-next-text-part 8-6(22)}, \text{int-ct-text-part 8-6(23)}

(87) \[ \text{ct-if-st}(t, cl) = \]
\begin{align*}
\text{is-<>}(cl) & \rightarrow \text{ct-update-cd}(t, cl) \\
\text{is-ct-if-st}(pst_t) & \rightarrow \text{ct-if-st}(pst_t, \text{tail}(cl)) \\
\end{align*}
\begin{align*}
\text{is-ct-p-if-st}(pst_t) & \rightarrow \text{ct-if-st}(pst_t, \text{tail}(cl)) \\
T & \rightarrow \text{ct-update-cd}(pst_t, \text{tail}(cl)) \\
\end{align*}

cont'd
where: \( \text{pst}_t = (\text{head}(cl) = 1 \to s-ct-prop-st \cdot s-ct-st \cdot s-ct-then(t), \)
\( \text{head}(cl) = 2 \to s-ct-prop-st \cdot s-ct-st \cdot s-ct-else(t) \)

for: is-ct-if-st(t) \lor is-ct-p-if-st(t)

\[(88)\]
\[
\text{ct-update-cd}(t, cl) = \\
\text{is-}(cl) \to \text{int-ct-prop-st}(t) \\
T \to s-ct-t; \\
s-ct-sc; \text{head}(cl) \\
s-ct-cd; \text{ct-stack-cd}(p) \\
s-ct-c; \text{ct-elem-execute}(cl)
\]

for: is-ct-prop-st(t) \lor is-ct-p-prop-st(t)

Ref.: int-ct-prop-st 8-6(24), ct-stack-cd 8-3(13)

Note: For each (procedure) text-part-list to be entered a new level of the control dump has to be established. (When entering an if statement a new level of \( \text{ctCD} \) is not necessary, because the then-component as well as the else-component represents only one statement).

8.3.7 If statement

The execution result is defined by a selection of one of the ct-statements found in the s-ct-then and s-ct-else component of the if statement. The selection is controlled by the bit-value of the expression specified in s-ct-decision of the if statement.

\[(89)\]
\[
\text{int-ct-if-st}(t) = \\
\text{int-ct-if}(r, s-ct-prop-st \cdot s-ct-st \cdot s-ct-then(t), \\
\text{ct-prop-st} \cdot s-ct-st \cdot s-ct-else(t)); \\
r: \text{ct-convert}(\text{BIT}, v); \\
v: \text{int-ct-expr}(s-ct-decision(t))
\]

for: is-ct-if-st(t) \lor is-ct-p-if-st(t)

Ref.: int-ct-expr 8-10(34), ct-convert 8-15(55)
8.3.8 Activate-, deactivate-, and declare statement

By these three ct-statements the control of the activation status of entry names and global variables is handled, which is hold in the s-ct-act component of their denotation entry in ctDN.

The first interpretation of a declare statement leads also to a change of the declare status from F to T for all identifiers specified in the declare statement, indicating that the identifiers may be used by means of other ct-statements, and that any further execution of that declare statement is equivalent with the execution of a null statement (cf. 7.4.2.6).

8.3.8 Activate-, deactivate-, and declare statement

(90) \[
\text{int-ct-if}(\text{test}, \text{then}, \text{else}) = \\
(\exists i)(\text{is-1-CHAR\-ct-value}(\text{BIT}, \text{elem}(i) \cdot s-ct-vr(t)) \leftarrow \\
\text{int-ct-prop-st}(\text{then}) \\
T \leftarrow \text{int-ct-prop-st}(\text{else})
\]

for : is-ct-string(test) & is-BIT s-ct-da(test)
(is-ct-prop-st \lor \text{is-ct-p-prop-st})(\text{then})
(is-ct-prop-st \lor \text{is-ct-p-prop-st})(\text{else})

Ref.: ct-value 2-2(6), \text{int-ct-prop-st} 8-6(24)

(91) \[
\text{int-ct-decl-st}(t) = \\
\text{ct-null};
\{\text{int-ct-decl-act}(\text{elem}(i) \cdot s-ct-id-list(t)) | 1 \leq i \leq \text{length}(s-ct-id-list(t))\}
\]

for : is-ct-declare-st(t)

Note: Declare statements appear only within text-part-lists.

(92) \[
\text{int-ct-decl-act}(\text{id}) =
\text{proper_id} \& \neg \text{s-ct-dcl}(\text{id}(ctE)(ctDN)) \leftarrow \\
\text{s-ct-dn}: \mu(\text{ctDN};<\text{s-ct-act}(\text{id}(ctE));T>, \text{<s-ct-dcl}(\text{id}(ctE));T>)
\text{proper_id} \& \text{s-ct-dcl}(\text{id}(ctE)(ctDN)) \leftarrow \text{ct-null}
T \leftarrow \text{error}
\]

where: \text{proper_id} = \neg \text{is-\-Q\-id}(ctE) \& (\text{is-ct-prop-var} \lor \text{s-ct-at}(\text{id}(ctE)(ctDN)) \lor \\
\text{is-ct-entry-decl} \lor \text{s-ct-at}(\text{id}(ctE)(ctDN)))

cont'd
for : is-id(id)

Note: The check done by proper id is necessary because regarding only the abstract syntax no connection between the declare statement and a declaration-part exists.

\[(93) \text{int-ct-act-st}(t) = \text{int-ct-act-deact}(s\text{-ct-id-list}(t), T)\]

for: is-ct-act-st(t)

\[(94) \text{int-ct-deact-st}(t) = \text{int-ct-act-deact}(s\text{-ct-id-list}(t), F)\]

for: is-ct-deact-st(t)

\[(95) \text{int-ct-act-deact}(\text{idl}, t) = \]

\[\text{is-\ensuremath{\llcorner}(idl) \rightarrow ct-null} \]

\[\text{is-\ensuremath{\llcorner}(ah) \& is-\ensuremath{\llcorner}(s\text{-ct-act\ensuremath{\llcorner}ah}(ctDN) \& is-LAB\text{-ct-at\ensuremath{\llcorner}ah}(ctDN) \& s-ct-dcl\ensuremath{\llcorner}ah}(ctDN) \rightarrow} \]

\[\text{int-ct-act-deact}(\text{tail(idl), t)}; \]

\[\text{ct-set-act(\text{head(idl), t)};} \]

\[\text{is-\ensuremath{\llcorner}(ah) \& head(idl) = mk-id(SUBSTR) \rightarrow} \]

\[\text{int-ct-act-deact}(\text{tail(idl), t)}; \]

\[\text{ct-set-act(\text{head(idl), t)};} \]

\[\text{update-g-cte(\text{head(idl), ct-af(*, head(idl)})}) \]

\[T \rightarrow \text{error}\]

where: \(ah = \text{head(idl)}(ctE)\)

for : is-id-list(idl), is-T(t) \& is-F(t)

Ref. : update-g-cte 8-19(69), ct-af 2-2(1), mk-id 2-3(9)

Note: If \(mk-id(SUBSTR)\) is specified in an activate or deactivate statement and has no entry in \(ctE\), then \(mk-id(SUBSTR)\) denotes per definition the built-in function and gets the proper entries in \(ctE\) and \(ctDN\) by update-g-cte (cf. 7.4.3.3).

\[(96) \text{ct-set-act}(\text{id}, t) = \]

\[s\text{-ct-dn: } \mu(\text{ctDN}; \text{<s\text{-ct-act\ensuremath{\llcorner}id}(ctE)>) : t})\]

for: is-id(id), is-T(t) \& is-F(t)

8.3.8
8.3.9 Include statement

\[(97) \quad \text{int-ct-include-st}(t) = \]
\[\quad \text{int-ct-include} (s-ct-id-list(t))\]
\[\text{for: is-ct-include-st}(t)\]

\[(98) \quad \text{int-ct-include}(idl) = \]
\[\quad \text{is-<>}(idl) \rightarrow \text{ct-null}\]
\[\quad T \rightarrow \text{int-ct-include}(\text{tail}(idl));\]
\[\quad \text{ct-program-text-execute}(\text{ct-sel}(\text{head}(idl)))(\text{ctET})\]
\[\text{for: is-ct-id-pair-list}(idl)\]
Ref.: \text{ct-program-text-execute} 8-3(12), ct-sel 2-3(7)
Note: ct-sel maps an identifier-pair onto a selector which specifies in the external text-storage ctET an external program that is executed in the following.

8.3.10 Return statement

\[(99) \quad \text{int-ct-return-st}(t) = \]
\[\quad \text{ct-exit};\]
\[\quad \text{ct-return}(r);\]
\[\quad r: \text{ct-convert}(s-ct-ret-type(\text{ctRI}), v);\]
\[\quad v: \text{int-ct-expr}(s-ct-expr(t))\]
\[\text{for: is-ct-return-st}(t)\]
Ref.: \text{int-ct-expr} 8-10(34), \text{ct-convert} 8-15(55)

\[(100) \quad \text{ct-return}(val) = \]
\[\quad s-ct-dn: \mu (\text{ctDN}; <s-ct-at>(s-ct-name(\text{ctRI})):s-ct-da(val)>,\]
\[\quad <s-ct-s>(s-ct-name(\text{ctRI})):s-ct-vr(val)>\]
\[\quad s-ct-p: \mu (\text{ctP}; <s-ct-old>(s-ct-pt(\text{ctRI})):F>\]
\[\text{for: is-ct-const}(val)\]
Note: The ct-constant, representing the converted value of the expression specified by the return statement, is stored in ctDN under a unique name which was generated when the function reference was encountered (int-ct-ref or int-ct-func) and which was stored into the s-ct-name component of ctRI when the procedure body was entered (int-ct-call). cont'd
The pointer found in the s-ct-cld component of ctRI allows the indication in the procedure body directory ctDN, that the execution of the procedure is completed and a new reference to it can be made.

\[(101)\]
\[
\text{ct-exit} =
\]
\[
\begin{align*}
\text{s-ct-e} &: \text{s-ct-e}(\text{ctD}) \\
\text{s-ct-ri} &: \text{s-ct-ri}(\text{ctD}) \\
\text{s-ct-d} &: \text{s-ct-d}(\text{ctD}) \\
\text{s-ct-t} &: \text{s-ct-t}(\text{ctD}) \\
\text{s-ct-sc} &: \text{s-ct-sc}(\text{ctD}) \\
\text{s-ct-cd} &: \text{s-ct-cd}(\text{ctD}) \\
\text{s-ct-c} &: \text{s-ct-c}(\text{ctD})
\end{align*}
\]

Note: The old state components, saved in the dump ctD when the procedure body was entered (int-ct-call), are reinstalled now, to control the continuation of the interpretation process.

8.3.11 Interpretation of the procedure-body

Function references encountered in ct-expressions as well as function references encountered within a ct-text refer to this section. In both cases the arguments of the reference are already evaluated, i.e. the argument list consists of addresses possessing a denotation.

\[(102)\]
\[
\text{int-ct-call}(\text{at}, \text{pt}, \text{agl}, \text{a}) =
\]
\[
\begin{align*}
\text{s-ct-e} &: \text{get-cte}(<\text{a}>) \\
\text{s-ct-d} &: \text{ctD}(<\text{s-ct-e} : \text{ctE}>, <\text{s-ct-ri} : \text{ctRI}>, <\text{s-ct-d} : \text{ctD}>, <\text{s-ct-t} : \text{ctT}>, <\text{s-ct-sc} : \text{ctSC}>, <\text{s-ct-cd} : \text{ctCD}>, <\text{s-ct-c} : \text{ctC}>) \\
\text{s-ct-p} &: \text{ctP}(<\text{s-ct-cld} : \text{T}>) \\
\text{s-ct-ri} &: \text{ctRI}(<\text{s-ct-name} : \text{a}>) \\
\text{s-ct-t} &: \text{ct-text-part-list}(<\text{bT}>) \\
\text{s-ct-sc} &: 0 \\
\text{s-ct-cd} &: \emptyset \\
\text{s-ct-c} &: \text{int-ct-next-text-part}; \\
\text{ct-check-ri}(\text{s-ct-ret-type}(\text{bT})); \\
\text{int-ct-p-decl-part}(\text{s-ct-decl-part}(\text{bT}), \text{s-ct-par-list}(\text{bT}, \text{agl})); \\
\text{update-p-cte}(\text{pt})
\end{align*}
\]
where: \( b_t = s\text{-}ct\text{-}body\circ pt(ctP) \)

for: is-ct-entry-decl(at), is-ad(pt), is-ad-list(agt), is-n(a)

Ref. : get-g-cte 8-19(67), int-ct-next-text-part 8-6(22)

Note: The function get-g-cte applied to the state \( \xi \) yields a selector which points to the outermost environment contained in \( \xi \), i.e. the environment where the procedure body was declared. The return information \( \text{ctRI} \) get the information which must be available during the interpretation of a return statement: the return-type, the pointer to the body (pt), and a unique name to which the function value must be assigned.

(103) \[
\text{update-p-cte}(pt) =
\text{ct-null};
\]

\[
\text{update-ct-id}(id, ct\text{-}af(pt, id)) | id \in ct\text{-}ident(s\text{-}ct\text{-}decl\text{-}part\circ s\text{-}ct\text{-}body\circ pt(ctP)) \land \\
\neg(\exists i)(id = elem(i) \circ s\text{-}ct\text{-}par\text{-}list\circ s\text{-}ct\text{-}body\circ pt(ctP))
\]

for: is-ad(pt)

Ref.: update-ct-id 8-4(16), ct-af 2-2(1), ct-ident 8-4(15)

Note: The environment is updated by all identifiers declared in the procedure-declaration-part which are not used as parameters. The necessary addresses are generated by ct-af using the selector pt which identifies the body in ctP as scope information.

(104) \[
\text{int-ct-p-decl-part}(t, pal, agt) =
\left(\forall i\right)i < \text{length}(pal) \Rightarrow elem(i, pal) \in ct\text{-}ident(t) \rightarrow
\]

\[
\text{int-ct-par-list}(t, pal, agt);
\left\{\text{int-ct-p-decl}(id(\text{ctE}), id(t)) | is-id(id) \land \neg is-\Omega \circ id(t) \land \\
\neg(\exists i)(id = elem(i, pal))\right\}
\]

T \rightarrow error

for: is-ct-p-decl-part(t), is-id-list(pal), is-ad-list(agt)

Ref.: ct-ident 8-4(15)

(105) \[
\text{int-ct-p-decl}(ad, at) =
\]

\[
is\text{-}LAB(at) \rightarrow s\text{-}ct\text{-}dn;\mu(\text{ctDN};<s\text{-}ct\text{-}at\circ ad\circ LAB>)
\]

\[
is\text{-}ct\text{-}prop\text{-}var(at) \rightarrow s\text{-}ct\text{-}dn;\mu(\text{ctDN};<s\text{-}ct\text{-}at\circ ad\circ at>, \\
<s\text{-}ct\text{-}dcl\circ ad\circ T>)
\]

for: is-ad(ad), is-LAB(at) \lor is-ct-prop-var(at)
(106) \( \text{int-ct-par-list}(t, \text{pal}, \text{agl}) = \) 

\[
(\forall i)(1 \leq \text{is-length}(\text{pal}) \supset \neg(\exists j)(1 \leq j \leq n) \land i \neq j \land \text{elem}(i, \text{pal}) = \text{elem}(j, \text{pal}) \land \text{ct-match}(s-\text{ct-at}(\text{elem}(i, \text{agl}, \text{ctDN})), \\
\text{elem}(i, \text{pal}, t)) \land \text{length}(\text{pal}) = \text{length}(\text{agl}) \rightarrow \\
\text{ct-null}; \\
\{ \text{update-ct-id}(\text{elem}(i, \text{pal}), \text{elem}(i, \text{agl})) \mid 1 \leq i \leq \text{length}(\text{pal}) \} \)
\]

\( T \rightarrow \text{error} \) for : is-ct-p-decl-part(\( t \)), is-id-list(\( \text{pal} \)), is-ad-list(\( \text{agl} \))
Ref.: ct-match 8-20(74), update-ct-id 8-4(16)

Note: At this point the connection of the parameters with their corresponding arguments is performed. The arguments have been evaluated at the time the function reference had been encountered and for each argument an address was passed indicating its entry in ctDN. This address now is connected with the corresponding parameter in an entry into the environment.

(107) \( \text{ct-check-ri}(\text{ri}) = \) 

\[ \text{ri} = \text{s-ct-ret-type} (\text{ctRI} ) \rightarrow \text{ct-null} \]

\( T \rightarrow \text{error} \) for: is-INTG(\( \text{ri} \)) \lor is-CHAR(\( \text{ri} \))

8.3.12 The scan and replacement mechanism for the ct-text

According to 8-6(22) and 8-6(23) the interpretation of a text-part starts with the interpretation of its ct-statement, and continues - if the normal flow of control is supposed - with the interpretation of the associated ct-text.

Metavariables

\( \text{ad} \quad \text{is-\*} \lor \text{is-n} \)

\( t \quad \text{is-char-val-list} \)

(108) \( \text{int-ct-text}(t, \text{ad}) = \) 

\[ \text{is-\text{< }>(t) } \rightarrow \text{ct-null} \]

\( T \rightarrow \text{int-ct-text}(r, \text{ad}); \)

\( r: \text{int-ct-tok}(t, \text{ad}) \) cont'd
Note: The second argument specifies whether a token which is not further replaceable is transferred into the result cell \(ctR\) (is-*\( (ad)\)) or is part of an argument of a function reference (is-n\( (ad)\)) which must be stored as an intermediate result in \(ctDN\) using the unique name \(ad\) as selector.

\[(109)\]

\[
\text{int-ct-tok}(t,\text{ad}) =
\]
\[
\text{is-ct-ident-head-ct-tok}(t) \rightarrow \text{int-ct-id}(t,\text{ad})
\]
\[
T \rightarrow \text{ct-copy}(t,\text{ad})
\]

\[(110)\]

\[
\text{is-ct-ident}(t) =
\]
\[
\text{is-letter}(\text{elem}(1,t)) \& (\forall i)(2\leq i\leq \text{length}(t) \land \text{is-alpham-char}(\text{elem}(i,t)))
\]

Note: This predicate constitutes a necessary condition for the replacement of a token.

\[(111)\]

\[
\text{ct-tok}(t) =
\]
\[
\text{ct-tok-1}(t,<<,\emptyset)
\]

\[(112)\]

\[
\text{ct-tok-1}(t,\text{token,ind}) =
\]
\[
\text{is-}<<(t) \& \text{is-}\emptyset(\text{ind}) \rightarrow <\text{token},>>
\]
\[
\text{is-}<<(t) \lor \text{is-}\text{COMMENT}(\text{ind}) \& \text{length}(t) < 2 \rightarrow \text{error}
\]
\[
\text{is-}\emptyset(\text{ind}) \& \text{is-letter-head}(t) \& ((\exists i)(\text{is-del-id}.) \& \text{is-}<<(\text{token}) \rightarrow
\]
\[
\begin{align*}
&\text{is-}<<(t) \land \text{length}(t) - \text{length}(\text{id}) \land \text{length}(\text{id}) - \text{length}(t) \\
&\text{is-}\text{LETTER}(\text{head}(t)) \& (\exists i)(\text{is-del-id}.) \& \text{is-}<<(\text{token}) \rightarrow
\end{align*}
\]
\[
\begin{align*}
&\text{ct-tok-1}(\text{tail} \cdot \text{tail}(t),\text{token} \land <\text{SLASH,ASTER}>,\text{COMMENT}) \\
&\text{is-}\text{COMMENT}(\text{ind}) \& \text{is-ASTER-head}(t) \& \text{is-SLASH-head}(t) \rightarrow
\]
\[
\begin{align*}
&\text{ct-tok-1}(\text{tail} \cdot \text{tail}(t),\text{token} \land <\text{ASTER,SLASH}>,\emptyset) \\
&T \rightarrow \text{ct-tok-1}(\text{tail}(t),\text{token} \land <\text{head}(t)>,\text{new}_{\text{ind}})
\end{align*}
\]

cont'd
where: \(\text{is-del-id}_i = (\text{is-delimiter}\_\text{elem}(i+1,t) \lor \text{is-\(\Omega\_\text{elem}(i+1,t)) \land (\forall k)(\text{is-elem}(i+1,t)) \lor (\forall k)(\text{is-\(\Omega\_\text{elem}(k,t))})
\]

\[
\text{lgth}_i = (\text{is-del-id}_i)
\]

\[
\text{new}_i = \text{is-\(\Omega\_\text{elem}(ind) \land \text{is-APOSTR}\_\text{head}(t) \rightarrow \text{STRING}
\]

\[
\text{is-STRING}(\text{ind}) \land \text{is-APOSTR}\_\text{head}(t) \rightarrow \Omega
\]

\[T \rightarrow \text{ind}
\]

for \(i\) is-char-val-list(token)

\[(\text{is-\(\Omega\_\text{STRING}\_\text{ind}) \land \text{is-APOSTR}\_\text{head}(t) \rightarrow \text{STRING}
\]

\[T \rightarrow \text{ind}
\]

Ref.: last-el 2-4(11)

Note: This function yields an ordered pair. The first element is the first "token" of the ct-text \(t\). The second element is the remaining ct-text. A "token" of \(t\) is either an identifier which - if it is surrounded by PL/I-characters within \(t\) - must be immediately surrounded by PL/I-delimiters, or any substring of the ct-text which is delimited by those identifiers.

The function also checks \(t\) for unmatched comment or character-string delimiters.

(113) \(\text{is-delimiter}(x) = \)

\[x \in \{\text{PLUS, MINUS, ASTER, SLASH, GT, EQ, LT, NOT, AND, OR, LEFT-PAR, RIGHT-PAR, COMMA, SEMIC, COLON, BLANK, APOSTR, POINT, PERC}\}
\]

(114) \(\text{ct-copy}(t, ad) = \)

\[
\text{is-\(*(ad) \rightarrow PASS}\_\text{tail}\_\text{ct-tok}(t)
\]

\[
\text{s-ct-r:ctR} \cap \text{head}\_\text{ct-tok}(t)
\]

\[T \rightarrow PASS}\_\text{tail}\_\text{ct-tok}(t)
\]

\[
\text{s-ct-dn:}\mu(\text{ctDN} ; s-ct-s\_ad:s-ct-s\_ad(\text{ctDN})) \cap \text{ct-representlist}(\text{CHAR}, \text{head}\_\text{ct-tok}(t)) >
\]

Ref.: ct-representlist 8-11(36)

Note: In the case that \(ad\) is a unique name the token has to be transformed to the proper value representation before it is stored into the entry of \(ad\) in \(\text{ctDN}\), where it is concatenated with the ct-\text{vr-list} which is already there.
where: $t_t = \text{tail}\cdot\text{ct-tok}(t)$

$$ah_t = \text{mk-id}\cdot\text{head}\cdot\text{ct-tok}(t)$$

$$dn_t = \text{mk-id}\cdot\text{head}\cdot\text{ct-tok}(t)\ (\text{ctE}\ (\text{ctDN}))$$

for: is-ct-ident(t)

Ref.: ct-convert 8-15(55), ct-pass 8-21(76), mk-id 2-3(9)

Note: The token representing a ct-identifier must be converted by means of the function mk-id to an object described by is-id. This object, the abstract form of the ct-identifier, gives access to the corresponding entry in ctE.
**ct-pass-text**(vr) =

```
PASS:<BLANK>\ct-valuelist(CHAR,s-ct-vr(vr))<BLANK>
```

for : is-ct-string(vr) & is-CHAR=ct-da(vr)

Ref.: ct-valuelist 8-11(37)

Note: Before the rescan can take place, the ct-string of type CHAR must be converted to a char-val-list, and to both ends of the list the abstract character value BLANK is appended, according to the rule that a substituted value must be surrounded by blanks.

**ct-arg-parse**(t) =

```
ct-arg-parse-1(<><>tail(t),O,F)
```

(117) **ct-arg-parse-1**(arg,argl,t,pcount,string) =

```
is-<(t)\error

is-RIGHT-PAR\head(t) & pcount=O & is-F(string) \rightarrow

<argl<arg>,tail(t)>
```

is-COMMA\head(t) & pcount=O & is-F(string) \rightarrow
t-ct-arg-parse-1(<>argl<arg>,tail(t),O,F)

T-ct-arg-parse-1(arg\head(t),argl,tail(t),upc_t,ust_t)

where: upc_t = is-LEFT-PAR\head(t) & is-F(string) → pcount + 1

is-RIGHT-PAR\head(t) & is-F(string) → pcount - 1

T → pcount

ust_t = is-APOST\head(t) → ¬string

T → string

for : is-char-val-list(arg)

is-char-val-list-list(argl)

is-intg-val(pcount)

(is-T & is-F)(string)

Note: This function yields an ordered pair. Its first element is a list whose elements are char-val-lists to be used as arguments of the function reference. The second element of the result is the remaining tail of the argument t.
\[(119) \text{int-ct-func}(at, id, al, ad) = \]
\[
\text{length}(al) = \text{length}\text{-s-ct-par-list}(at) \implies \\
\text{ct-copy-ret}(t, ad) ; \\
t : \text{ct-function-val}(a) ; \\
\text{int-ct-call}(at, s-ct-s(id(\text{ctE})(\text{ctDN})), agl, a) ; \\
a : \text{ct-un-name} , \\
agl : \text{ct-dummy-assign-list}(id, argl) ; \\
\text{argl : int-ct-arg-func-list}(al, s-ct-par-list(at), <>)
\]

\[
T \implies \text{error}
\]

for : is-ct-entry-decl(at), is-id(id), is-char-val-list-list(al)

Ref.: \text{ct-dummy-assign-list} 8-21(75), \text{ct-un-name} 8-21(78), \\
\text{int-ct-call} 8-29(102), \text{ct-function-val} 8-22(79)

Note: On the one hand the unique name a is passed to \text{int-ct-call} which stores
a into ctRI (necessary for the interpretation of the return statement),
on the other hand the unique name is passed to \text{ct-function-val}, giving
access to the value of the function procedure when the interpretation
of the body is terminated.

\[(120) \text{int-ct-arg-func-list}(agl, pal, r) = \]
\[
\text{is-<>}(agl) \implies \text{PASS: r} \\
T \implies \text{int-ct-arg-func-list}(\text{tail}(agl), \text{tail}(pal), v) ; \\
v : \text{ct-pass-cat}(r, a) ; \\
a : \text{ct-func-arg}(\text{head}(agl), \text{head}(pal))
\]

for : is-char-val-list-list(agl), is-ct-descr-list(pal),
is-ct-const-list(r)

Ref.: \text{ct-pass-cat} 8-20(73)

\[(121) \text{ct-func-arg}(ag, pa) = \]
\[
\text{ct-convert}(\text{ct-pat}(v, pa), v) ; \\
v : \text{ct-pass-res}(\text{CHAR}, r) ; \\
r : \text{int-ct-arg-text}(ag)
\]

for : is-char-val-list(ag), is-ct-descr(pa)

Ref.: \text{ct-pass-res} 8-12(41), \text{ct-convert} 8-15(55)
Note: Each char-val-list representing an argument is rescaned for possible replacement using the instruction int-ct-text. At that point it is necessary to indicate by the second argument of int-ct-text that the result of the replacement process must not be transferred into \( \text{ctR} \), but is stored as an intermediate result into the \( \text{ctDN} \) using the unique name \( \text{ad} \) as selector.

\[
\text{ct-pat}(t,p) = \begin{align*}
\text{is}^-*(p) &\rightarrow s\text{-ct-da}(t) \\
T &\rightarrow p
\end{align*}
\]

for: is-ct-const\((t)\), is-ct-descr\((p)\)

\[
\text{ct-arg-eml}(n) = \begin{align*}
s\text{-ct-dn} &\equiv \mu(\text{ctDN};<s\text{-ct-son:<}>)
\end{align*}
\]

for: is-n\((n)\)

\[
\text{restore-ct-arg-text}(n) = \begin{align*}
\text{PASS} &\equiv s\text{-ct-son}(\text{ctDN})
\end{align*}
\]

for: is-n\((n)\)

\[
\text{ct-copy-ret}(vr,ad) = \begin{align*}
\text{int-ct-text}(v,ad); \\
v &\equiv \text{ct-pass-text}(r); \\
r &\equiv \text{ct-convert}(\text{CHAR},vr)
\end{align*}
\]

for: is-ct-const\((vr)\)

Note: The value returned by the interpretation of a function gets the abstract character value BLANK appended to each end and the rescan and replacement mechanism is reapplied.
(127) \text{int-ct-built-in-func}(id, al, ad) =
\begin{align*}
\text{id} & = \text{mk-id}(\text{SUBSTR}) \& 2 \leq \text{length}(al) \leq 3 \\
\text{ct-copy-ret}(t, ad);
\end{align*}
\begin{align*}
t & : \text{int-ct-built-in}(id, agl) \\
agl & : \text{ct-dummy-assign-list}(id, argl) \\
argl & : \text{int-ct-arg-func-list}(al, \text{ct-built-in-par}(id), <>)
\end{align*}
\begin{align*}
T & \rightarrow \text{error}
\end{align*}
\text{for : is-char-val-list-list}(al) \\
\text{Ref. : ct-dummy-assign-list 8-21(75), int-ct-built-in 8-22(80), mk-id 2-3(9)}
This index lists all names used in the document, with the exception of:

names defined in chapter 3 (concrete syntax),
names of elementary objects,
names of abbreviations and meta-variables,
names of the form is-[name] or is-c-[name] where [name] is a string of capital letters.

In general the names are referenced by the form X-YY(ZZZ), where X is the number of the main chapter, YY is the page number within the main chapter and ZZZ is the number of a definition within the main chapter.

The following conventions hold:

(1) Names defined in section 1.4 of /4/ or in chapter 1 of /6/ which are used in the present document are listed with the entry (/4/) or (/6/) respectively.

(2) For names defined in section 2.2 only the reference to the definition is given.

(3) For all other names all instances of use in a formula or in an abbreviation are given. Multiple occurrences within a formula are indicated by multiple references to the formula. The definition of a name is indicated by an underlined reference.

(4) Occurrences of names of the form is-pred-list are listed under the entry is-pred.
abs ........................................ 8-16(62), 8-16(62), 8-17(63)
char-val ........................................ 
collect ........................................ 8-14(50), 8-14(50), 8-14(51), 8-16(62), 8-16(62)
conc-tp ........................................ 
ct-abs-div ........................................ 
ct-af ........................................ 2-1(1), 2-4(14), 2-5(21), 2-18(56), 2-21(75), 8-27(95), 8-30(103)
ct-arg-eql ........................................ 8-35(117), 8-35(118), 8-35(118), 8-35(119)
ct-arg-parse ........................................ 
ct-arg-parse-1 ........................................ 
ct-arith-comp ........................................ 8-12(42), 8-12(43), 8-13(41)
ct-arith-res ........................................ 8-11(40), 8-13(40)
ct-arith-val ........................................ 8-13(49), 8-13(49), 8-13(49), 8-14(59)
ct-assign ........................................ 8-7(25), 8-9(30), 8-9(32), 8-10(33)
ct-bit-res ........................................ 8-11(40), 8-13(46), 8-13(46), 8-13(46), 8-13(46)
ct-bit-test ........................................ 8-20(71), 8-20(72)
ct-bit-to-int ........................................ 8-15(55), 8-16(59)
ct-bit-val ........................................ 8-16(59), 8-16(59), 8-16(59), 8-16(60), 8-16(60)
ct-blank-fill ........................................ 8-16(61), 8-16(61), 8-17(64), 8-17(64)
ct-built-in-par ........................................ 8-18(66), 8-19(66), 8-38(127)
ct-char-to-bit ........................................ 8-15(55), 8-17(65), 8-17(65)
ct-char-to-int ........................................ 8-15(55), 8-15(56), 8-15(56)
ct-check-fil ........................................ 8-20(71), 8-31(101)
ct-collat ........................................ 2-1(2), 8-12(44), 8-12(44)
ct-collect ........................................ 8-2(31), 8-21(42), 8-22(82)
ct-comp-res ........................................ 8-11(40), 8-12(42)
ct-cond-expr ........................................ 8-8(27), 8-8(29), 8-9(30)
ct-const-test ........................................ 8-10(59), 8-10(59)
ct-convert 8-7(25), 8-9(30), 8-9(32), 8-11(38), 8-11(38), 8-14(52), 8-15(55), 8-20(71), 8-25(89), 8-28(93), 8-34(115), 8-36(121), 8-37(126)
ct-copy ........................................ 8-32(109), 8-33(114), 8-34(115)
| ct-copy-set     | 8-36(119), 8-37(126), 8-38(127) |
| ct-div         | 8-14(49), 8-14(50) |
| ct-dummy-assign| 8-21(75), 8-21(77) |
| ct-dummy-assign-list | 8-18(66), 8-18(66), 8-21(75), 8-36(119), 8-38(127) |
| ct-elem-execute| 8-24(85), 8-24(85), 8-25(88) |
| ct-exit        | 8-28(99), 8-29(101) |
| ct-func-arg    | 8-18(66), 8-22(79), 8-36(119) |
| ct-ident       | 8-4(14), 8-4(14), 8-4(15), 8-4(19), 8-30(103), 8-30(104) |
| ct-if-st       | 8-24(86), 8-24(87), 8-24(87) |
| ct-infix-int   | 8-11(38), 8-11(49) |
| ct-initial-state|                                                                 |
| ct-int-to-char | 8-15(55), 8-16(61) |
| ct-int-to-string| 8-15(55), 8-16(61), 8-16(62), 8-16(62) |
| ct-labelf      | 8-23(83), 8-23(84), 8-23(84), 8-23(84), 8-23(84), 8-23(84), 8-23(84), 8-23(84) |
| ct-log-string  | 8-13(46), 8-13(46), 8-13(47) |
| ct-match       | 8-20(71), 8-20(74), 8-31(106) |
| ct-not-val     | 8-14(53), 8-14(54), 8-14(54), 8-14(54) |
| ct-null        | 8-4(14), 8-4(17), 8-4(18), 8-6(21), 8-6(24), 8-9(30), 8-19(69), 8-26(91), 8-26(92), 8-27(95), 8-28(98), 8-30(103), 8-31(106), 8-31(107), 8-31(108) |
| ct-parse       | 4-2(1) |
| ct-parse-48    | 4-4(10) |
| ct-pass        | 8-21(75), 8-21(76), 8-34(115), 8-34(115), 8-34(115), 8-34(115) |
| ct-pass-cat    | 8-19(70), 8-20(73), 8-36(120) |
| ct-pass-int    | 8-11(40), 8-11(40), 8-11(40), 8-11(40), 8-12(41), 8-14(53), 8-14(53), 8-15(55), 8-15(55), 8-15(55), 8-15(55), 8-15(55), 8-15(55), 8-22(80), 8-22(80), 8-36(121) |
| ct-pass-text   | 8-34(115), 8-35(116), 8-37(126) |
| ct-pat         | 8-20(71), 8-36(121), 8-37(123) |
headlist       2-4(101), 8-16(56), 8-16(57), 8-16(57), 8-16(58), 8-16(60)
idp-set       5-8(29), 5-8(29), 5-8(29)
insert-space   4-2(2), 4-3(5), 4-3(5), 4-3(5), 4-3(5), 4-3(5), 4-3(5), 4-3(5)
insert-space-4B 4-4(11), 4-6(16), 4-6(16), 4-6(16), 4-6(16), 4-6(16), 4-6(16)
int-ct-act-deact 8-27(93), 8-27(94), 8-27(95), 8-27(95), 8-27(95), 8-27(95)
int-ct-act-st   8-6(24), 8-27(95)
int-ct-arg     8-19(70), 8-29(71)
int-ct-arg-func-list 8-36(119), 8-36(120), 8-36(120), 8-36(127)
int-ct-arg-list 8-18(66), 8-18(66), 8-19(70), 8-19(70)
int-ct-arg-text 8-36(121), 8-37(122)
int-ct-assign-st 8-6(24), 8-9(32)
int-ct-built-in 8-18(66), 8-22(80), 8-38(127)
int-ct-built-in-func 8-34(115), 8-38(127)
int-ct-byte-exp 8-7(25), 8-7(25), 8-7(26)
int-ct-call     8-19(66), 8-29(102), 8-36(119)
int-ct-deact-st 8-6(24), 8-27(94)
int-ct-decl     8-4(18), 8-4(19)
int-ct-decl-act 8-26(91), 8-26(92)
int-ct-decl-part 8-3(12), 8-4(18)
int-ct-decl-st  8-6(24), 8-29(111)
int-ct-deo      8-7(25), 8-8(27)
int-ct-expr     8-7(25), 8-7(26), 8-8(27), 8-9(30), 8-9(30), 8-9(32), 8-10(34), 8-10(34), 8-10(34), 8-10(34), 8-20(71), 8-25(89), 8-28(99)
int-ct-func     8-34(115), 8-34(115), 8-36(119)
int-ct-goto     8-23(83), 8-24(105)
int-ct-goto-st  8-6(24), 8-23(83), 8-24(105)
int-ct-group    8-6(24), 8-6(24), 8-7(25)
int-ct-group-text 8-5(24), 8-8(27), 8-9(30), 8-9(31)
int-ct-id       8-32(109), 8-34(115)
int-ct-if       8-25(89), 8-26(90)
int-ct-if-st

int-ct-include

int-ct-include-st

int-ct-infix-exp

int-ct-next-text-part

int-ct-p-decl

int-ct-p-decl-part

int-ct-par-list

int-ct-prefix-exp

int-ct-prop-st

int-ct-ref

int-ct-return-st

int-ct-text

int-ct-text-part

int-ct-tok

intg-val

is-ad

is-alpham-char

is-attr-of

is-c-ct-activate-statement

is-c-ct-alphabetic-character

is-c-ct-assignment-statement

is-c-ct-attribute

is-c-ct-balanced-statement

is-c-ct-bit

is-c-ct-bit-string

is-c-ct-character-string

is-c-ct-comment

is-c-ct-comment-symbol
<table>
<thead>
<tr>
<th>is-c-ct-sentence</th>
<th>4-8(1), 4-8(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>is-c-ct-simple-group</td>
<td>4-2(4), 4-5(13), 4-10(11), 4-10(11), 5-8(31), 5-9(34)</td>
</tr>
<tr>
<td>is-c-ct-simple-reference</td>
<td>4-11(14), 4-11(15), 4-11(16)</td>
</tr>
<tr>
<td>is-c-ct-space</td>
<td>4-3(5), 4-3(5), 4-4(9), 4-6(16), 4-6(16)</td>
</tr>
<tr>
<td>is-c-ct-statement</td>
<td>4-8(2), 4-8(31), 4-9(4), 4-9(4)</td>
</tr>
<tr>
<td>is-c-ct-string-character</td>
<td>4-19(69), 4-19(70)</td>
</tr>
<tr>
<td>is-c-ct-string-part</td>
<td>4-19(71), 4-19(73)</td>
</tr>
<tr>
<td>is-c-ct-string-part-char</td>
<td>4-19(73), 4-20(74)</td>
</tr>
<tr>
<td>is-c-ct-text</td>
<td>4-2(3), 4-5(12), 4-8(1), 4-8(1), 4-19(71), 5-7(27)</td>
</tr>
<tr>
<td>is-c-ct-text-character</td>
<td>4-19(71), 4-19(72)</td>
</tr>
<tr>
<td>is-c-ct-unconditional-statement</td>
<td>Note of 6-5(50), 4-19(70), 4-20(74), 4-20(76)</td>
</tr>
<tr>
<td>is-char-val</td>
<td>4-3(6), 6-2(18), 6-4(46), 8-1(1)</td>
</tr>
<tr>
<td>is-ct-act-st</td>
<td>6-2(20), 6-4(42), 8-6(24)</td>
</tr>
<tr>
<td>is-ct-arith-op</td>
<td>6-3(27), 6-3(39), 8-11(39), 8-11(40)</td>
</tr>
<tr>
<td>is-ct-assign-st</td>
<td>6-2(13), 6-2(20), 6-3(42), 8-6(24)</td>
</tr>
<tr>
<td>is-ct-bit-op</td>
<td>6-3(27), 6-3(28), 8-11(39), 8-11(40)</td>
</tr>
<tr>
<td>is-ct-body</td>
<td>6-1(5), 6-1(5), 6-1(81), 8-1(5)</td>
</tr>
<tr>
<td>is-ct-c</td>
<td>7-5(9), 8-1(1), 8-2(7), 8-2(9)</td>
</tr>
<tr>
<td>is-ct-cd</td>
<td>8-1(1), 8-2(7), 8-2(7), 8-2(9)</td>
</tr>
<tr>
<td>is-ct-comp-op</td>
<td>6-3(27), 6-3(29), 8-11(39), 8-11(39), 8-11(39), 8-11(40)</td>
</tr>
<tr>
<td>is-ct-const</td>
<td>6-3(25), 6-3(35), 9-10(34)</td>
</tr>
<tr>
<td>is-ct-d</td>
<td>8-1(1), 8-2(7), 8-2(9)</td>
</tr>
<tr>
<td>is-ct-deact-st</td>
<td>6-2(20), 6-4(42), 8-6(24)</td>
</tr>
<tr>
<td>is-ct-decl</td>
<td>6-1(2), 6-1(31)</td>
</tr>
<tr>
<td>is-ct-decl-part</td>
<td>6-1(2), 6-1(31)</td>
</tr>
<tr>
<td>is-ct-declare-st</td>
<td>6-2(20), 6-4(38), 8-6(24)</td>
</tr>
<tr>
<td>is-ct-descr</td>
<td>6-1(6), 6-1(71)</td>
</tr>
<tr>
<td>is-ct-dn</td>
<td>8-1(1), 8-1(4)</td>
</tr>
<tr>
<td>is-ct-e</td>
<td>8-1(1), 8-1(31), 8-2(9)</td>
</tr>
<tr>
<td>is-ct-entry</td>
<td>6-1 (3), 6-1 (5), 6-1 (6), 8-1 (4), 8-4 (14), 8-18 (66), 8-26 (92), 8-34 (115)</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>is-ct-entry-decl</td>
<td>6-1 (5), 6-1 (5), 6-1 (6), 8-1 (4), 8-4 (14), 8-18 (66), 8-26 (92), 8-34 (115)</td>
</tr>
<tr>
<td>is-ct-et</td>
<td>8-1 (1), 8-1 (11), 8-3 (11)</td>
</tr>
<tr>
<td>is-ct-expr</td>
<td>6-2 (15), 6-2 (15), 6-2 (17), 6-3 (22), 6-3 (24), 8-3 (25), 6-3 (26), 6-3 (31), 6-3 (33), 6-3 (34), 6-4 (41), 8-20 (71)</td>
</tr>
<tr>
<td>is-ct-goto-st</td>
<td>6-2 (13), 6-2 (20), 6-4 (39), 8-6 (24)</td>
</tr>
<tr>
<td>is-ct-group</td>
<td>6-2 (20), 6-4 (21), 8-6 (24)</td>
</tr>
<tr>
<td>is-ct-id-pair</td>
<td>6-4 (44), 6-4 (44), 8-1 (2)</td>
</tr>
<tr>
<td>is-ct-ident</td>
<td>8-32 (109), 8-32 (110)</td>
</tr>
<tr>
<td>is-ct-if-st</td>
<td>6-2 (20), 6-3 (22), 6-3 (22), 6-3 (23)</td>
</tr>
<tr>
<td>is-ct-if-text-part</td>
<td>6-3 (22), 6-3 (22), 6-3 (22)</td>
</tr>
<tr>
<td>is-ct-include-st</td>
<td>6-2 (20), 6-4 (44), 8-6 (24)</td>
</tr>
<tr>
<td>is-ct-infix-exp</td>
<td>6-3 (25), 6-4 (26), 8-10 (34)</td>
</tr>
<tr>
<td>is-ct-infix-op</td>
<td>6-3 (26), 6-3 (27)</td>
</tr>
<tr>
<td>is-ct-intq</td>
<td>6-3 (35), 6-3 (36)</td>
</tr>
<tr>
<td>is-ct-iteration</td>
<td>6-2 (14), 6-2 (12), 6-2 (21)</td>
</tr>
<tr>
<td>is-ct-null-st</td>
<td>6-2 (13), 6-2 (20), 6-4 (49), 8-6 (24)</td>
</tr>
<tr>
<td>is-ct-p</td>
<td>8-1 (1), 8-1 (5)</td>
</tr>
<tr>
<td>is-ct-p-decl</td>
<td>6-1 (9), 6-1 (10)</td>
</tr>
<tr>
<td>is-ct-p-decl-part</td>
<td>6-1 (8), 6-1 (9)</td>
</tr>
<tr>
<td>is-ct-p-group</td>
<td>6-2 (13), 6-2 (14), 8-6 (24)</td>
</tr>
<tr>
<td>is-ct-p-if-st</td>
<td>6-2 (13), 6-2 (17), 8-6 (24), 8-23 (84), 8-23 (84), 8-24 (86), 8-24 (87)</td>
</tr>
<tr>
<td>is-ct-p-prop-st</td>
<td>6-2 (12), 6-2 (13)</td>
</tr>
<tr>
<td>is-ct-p-st</td>
<td>6-2 (11), 6-2 (12)</td>
</tr>
<tr>
<td>is-ct-p-text-part</td>
<td>6-1 (8), 6-2 (11), 6-2 (13), 6-2 (14), 6-2 (17), 6-2 (17), 6-2 (17), 6-2 (20), 6-2 (20)</td>
</tr>
<tr>
<td>is-ct-paren-exp</td>
<td>6-3 (25), 6-3 (33), 8-10 (34)</td>
</tr>
<tr>
<td>is-ct-prefix-exp</td>
<td>6-3 (25), 6-3 (33), 8-10 (34)</td>
</tr>
<tr>
<td>is-ct-prefix-op</td>
<td>6-3 (31), 6-3 (32)</td>
</tr>
<tr>
<td>is-ct-program</td>
<td>6-1 (1), 6-1 (2), 6-1 (2)</td>
</tr>
<tr>
<td>is-ct-prop-st</td>
<td>6-2 (19), 6-2 (20)</td>
</tr>
<tr>
<td>Token Type</td>
<td>Entries</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>is-ct-prop-var</td>
<td>6-1(3), 6-1(4), 6-1(10), 8-1(4), 8-4(19), 8-7(25), 8-9(32), 8-18(66), 9-20(71), 8-26(92), 8-30(105), 8-34(115)</td>
</tr>
<tr>
<td>is-ct-ref</td>
<td>6-3(25), 6-3(34), 8-10(34)</td>
</tr>
<tr>
<td>is-ct-return-st</td>
<td>6-2(13), 6-9(41), 8-6(24)</td>
</tr>
<tr>
<td>is-ct-ri</td>
<td>8-1(1), 8-2(18), 8-2(9)</td>
</tr>
<tr>
<td>is-ct-simple-var</td>
<td>6-2(15), 6-2(16), 6-3(24), 8-18(66), 8-20(71)</td>
</tr>
<tr>
<td>is-ct-st</td>
<td>6-2(18), 6-2(19), 6-3(23)</td>
</tr>
<tr>
<td>is-ct-state</td>
<td>8-1(11)</td>
</tr>
<tr>
<td>is-ct-string</td>
<td>6-3(35), 6-4(37), 8-12(42)</td>
</tr>
<tr>
<td>is-ct-t</td>
<td>8-1(1), 8-2(16), 8-2(7), 8-2(9)</td>
</tr>
<tr>
<td>is-ct-text-part</td>
<td>6-1(1), 6-2(18), 6-2(20), 6-2(21), 8-2(6), 8-6(24), 8-23(84)</td>
</tr>
<tr>
<td>is-ct-vr</td>
<td>2-4(13)</td>
</tr>
<tr>
<td>is-decl</td>
<td>5-2(3), 5-2(4), 5-3(10), 5-3(10)</td>
</tr>
<tr>
<td>is-del-char</td>
<td>6-4(46), 6-5(50)</td>
</tr>
<tr>
<td>is-delimiter</td>
<td>8-32(112), 8-32(112), 8-33(113)</td>
</tr>
<tr>
<td>is-digit</td>
<td>6-4(47), 6-4(49), 8-15(56), 8-16(57), 8-16(58)</td>
</tr>
<tr>
<td>is-entry-cont</td>
<td>5-2(3), 5-2(4), 5-3(9), 5-3(10), 5-3(11), 5-3(11), 5-4(12), 5-4(12), 5-4(12)</td>
</tr>
<tr>
<td>is-entry-decl-cont</td>
<td>5-2(3), 5-2(4), 5-2(5), 5-3(10), 5-3(11)</td>
</tr>
<tr>
<td>is-extralingual-char</td>
<td>Note of 6-5(50), 6-5(50)</td>
</tr>
<tr>
<td>is-id</td>
<td>2-4(16)</td>
</tr>
<tr>
<td>is-int-val</td>
<td>2-4(12)</td>
</tr>
<tr>
<td>is-label-cont</td>
<td>5-2(4), 5-3(11)</td>
</tr>
<tr>
<td>is-letter</td>
<td>6-4(47), 6-4(49), 8-32(110), 8-32(112), 8-32(112)</td>
</tr>
<tr>
<td>is-list</td>
<td>(/6/)</td>
</tr>
<tr>
<td>is-local-to</td>
<td>5-2(3), 5-2(3), 5-2(3), 5-3(19)</td>
</tr>
<tr>
<td>is-mult-decl</td>
<td>5-2(3), 5-3(19)</td>
</tr>
<tr>
<td>is-n</td>
<td>2-4(15)</td>
</tr>
<tr>
<td>is-var-decl-cont</td>
<td>5-2(4), 5-2(16), 5-3(11)</td>
</tr>
<tr>
<td>iterate-ct-do</td>
<td>8-8(27), 8-9(30)</td>
</tr>
<tr>
<td>last-el</td>
<td>2-4(11), 8-15(56), 8-15(56), 8-16(57), 8-16(57), 8-16(58), 8-16(58), 8-16(60), 8-32(112)</td>
</tr>
</tbody>
</table>
restore-ct-arg-text

s

sign

slen gth

tail

trans-const

trans-do-spec

trans-else-p-st

trans-else-st

trans-end-clause

trans-expr

trans-group

trans-if-st

trans-infix

trans-label-list

trans-lib-spec

trans-p-end-clause

trans-p-group

trans-p-if-st

trans-p-selist

trans-p-selist-1

trans-p-st

trans-prop-st

trans-se

trans-siref

trans-st

update-ct-id

update-ct-stat

update-ct-te
update-cter 8-4 (19) , 8-4 (19) , 8-5 (21)

update-gcpte 8-18 (66) , 8-19 (63) , 8-27 (95)

update-pcpte 8-29 (102) , 8-30 (103)
APPENDIX II: LIST OF ELEMENTARY OBJECTS

This index lists all explicitly named elementary objects used in the document, with the exception of selector-names.

| {} | D-CHAR | LEFT-PAR | SLASH |
| *  | DCL    | LT       | STRING |
| <> | DEACT  | M-CHAR   | SUBSTR |
| #  | DIV    | MINUS    | T      |
| A-CHAR | DOLLAR | MULT     | T-CHAR |
| ACT | E-CHAR | N-CHAR   | U-CHAR |
| ADD | EQ     | NE       | V-CHAR |
| AND | F      | NOT      | W-CHAR |
| APOSTR | F-CHAR | NULL     | X-CHAR |
| ASSIGN | G-CHAR | NUMBER-SIGN | Y-CHAR |
| ASTER | GE     | O-CHAR   | 2-CHAR |
| B-CHAR | GOTO   | OR       | 0-CHAR |
| BIT | GT     | P-CHAR   | 1-CHAR |
| BLANK | H-CHAR | PERC     | 2-CHAR |
| BREAK | I-CHAR | PLUS     | 3-CHAR |
| BUILTIN | IF     | POINT    | 4-CHAR |
| C-CHAR | INCL   | Q-CHAR   | 5-CHAR |
| CAT | INTG   | QUEST    | 6-CHAR |
| CHAR | J-CHAR | R-CHAR   | 7-CHAR |
| COLON | K-CHAR | RETURN   | 8-CHAR |
| COMM-AT | L-CHAR | RIGHT-PAR | 9-CHAR |
| COMMA | LAB    | S-CHAR   |       |
| COMMENT | LE     | SEMIC    |       |