This Update provides replacement pages for the Abstract Syntax and Interpretation of PL/I (TR 25.098). The location and extent of the changes are as follows:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page(s)</th>
<th>Replacements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5-8</td>
<td>5-6, 6.1, 7, 8</td>
</tr>
<tr>
<td>3</td>
<td>9-22</td>
<td>9-16, 16.1, 17-18, 18.1, 19-22</td>
</tr>
<tr>
<td></td>
<td>31-32</td>
<td>31-32</td>
</tr>
<tr>
<td>7</td>
<td>3-4</td>
<td>3-4</td>
</tr>
<tr>
<td></td>
<td>7-10</td>
<td>7-10, 10-1</td>
</tr>
<tr>
<td></td>
<td>13-16</td>
<td>13-14, 14.1, 15, 16</td>
</tr>
<tr>
<td>8</td>
<td>5-6</td>
<td>5-6</td>
</tr>
<tr>
<td></td>
<td>9-12</td>
<td>9-12</td>
</tr>
<tr>
<td></td>
<td>15-18</td>
<td>15-18</td>
</tr>
<tr>
<td></td>
<td>21-22</td>
<td>21-22</td>
</tr>
<tr>
<td></td>
<td>25-28</td>
<td>25-28, 28.1</td>
</tr>
<tr>
<td></td>
<td>31-32</td>
<td>31-32</td>
</tr>
<tr>
<td>9</td>
<td>7-10</td>
<td>7-10</td>
</tr>
<tr>
<td></td>
<td>33-34</td>
<td>33-34, 34.1</td>
</tr>
<tr>
<td>10</td>
<td>3-10</td>
<td>3-8, 8.1, 9, 10, 10.1</td>
</tr>
<tr>
<td></td>
<td>15-18</td>
<td>15, 16, 17, 18</td>
</tr>
<tr>
<td>11</td>
<td>57-58</td>
<td>57-58</td>
</tr>
</tbody>
</table>

A change is indicated by a vertical line to the left of the change.

Because this Update does not include a new Cross-Reference Index, there follows a list of new formulas introduced and formulas moved from one part of the document to another. The new formulas include those replacing deleted ones.

1-7 (5.1) 'is-ident' moved from 3-18(69)
'aggr-var' appears instead of 'aggr-entry' in 8-26(70.2)
'aggr-scalar-ref' appears instead of 'scalar-entry' in 8-26(70.3)

The main changes are to Chapters 3, 7, 8, 9 and 10. Those made in Chapters 1 and 11 are mainly to ensure internal consistency.

This Update was prepared by the Language Control and Definition Group at the Hursley Laboratory with assistance from IBM Laboratory, Vienna.

File this cover letter at the back of the index to provide a record of changes.

IBM United Kingdom Laboratories Ltd., Language Control and Definition Group, Hursley Park, Winchester, Hampshire, England
\[ \mu(x;\{\}) = x \]

3) \[ \mu_0(<a_1:x_1>,<a_2:x_2>,\ldots,<a_n:x_n>) \]

   The meaning of \( \mu_0 \) is defined by the following equation:
   \[ \mu_0(<a_1:x_1>,<a_2:x_2>,\ldots,<a_n:x_n>) = \mu(\mu_0(<a_1:x_1>,<a_2:x_2>,\ldots,<a_n:x_n>)) \]

4) \[ \mu_0(\{<c:x> | p(c,x)\}) \]

   The above form is analogous to 2), it is defined by:
   \[ \mu_0(\{<c:x> | p(c,x)\}) = \mu(\mu_0(\{<c:x> | p(c,x)\})) \]

5) \[ \delta(x;a_1,a_2,\ldots,a_n) \]

   This function deletes the \( a_i \) components from \( x \), it is defined by the following equation:
   \[ \delta(x;a_1,a_2,\ldots,a_n) = \mu(x;a_1:a_2:\ldots:a_n) \]

6) \[ \delta(x;\{s | p(s)\}) \]

   The above form is analogous to 2), it is defined by:
   \[ \delta(x;\{s | p(s)\}) = \mu(x;\{s \}) \]

The operators \( \mu, \mu_0, \delta \) and the application of selectors to objects can be used to form expressions denoting objects. One may write in the previous forms expressions denoting objects in places where an object is required, and expressions denoting selectors in places where a selector is required. Functions may be defined whose domains and/or ranges are objects. This applies also to predicates, in which case the range is restricted to the truth values.

1.1.3.4 The class of objects used for the formal definition of PL/I

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

The following simple selectors are used:

1) selectors denoted by strings of small letters, digits, hyphens, prefixed by \( s- \)

2) the range of the following functions
   \[ \text{sel(idl)} \] for is-id-list(idl)
   \[ s(i) \] for is-intg-val(i)
   \[ mk-id(cl) \] for is-char-val-list(cl)
   \[ elem(i) \] for is-intg-val(i)

The ranges of these one to one functions have no elements in common and do not contain selectors mentioned under 1), 3) and 4).

3) attention identifications, i.e. the set of elements satisfying the predicate \( \text{is-ident} \)

4) data set names, i.e. the set of elements satisfying the predicate \( \text{is-ds-n} \)
From these selectors the semigroup of all composite selectors is formed, including the identity function \( I \), which is the unity with respect to functional composition.

Composite selectors satisfy the predicate:

\[
(1) \quad \text{is-selector} = \]

The following elementary objects are used:

1) objects denoted by strings of capital letters, hyphens and digits without definition place and the special objects *, <>, [ ]
2) numbers, satisfying the predicate \( \text{is-num-val} \), represented as decimal numbers
3) character values, satisfying the predicate \( \text{is-char-val} \), but not mentioned under 1) (extralingual characters)
4) composite selectors
5) identifiers, satisfying the predicate \( \text{is-id} \)
6) unique names, satisfying the predicate \( \text{is-n} \)
7) value representations, satisfying the predicate \( \text{is-vr} \)
8) data sets, satisfying the predicate \( \text{is-ds} \)
9) sizes of storage, satisfying the predicate \( \text{is-size} \)
10) pointer values, satisfying the predicate \( \text{is-ptr-val} \)
11) finite sets of objects

All objects satisfy the predicate:

\[
(2) \quad \text{is-ob}(x) = \top
\]

The predicate \( \text{is-} \) is only satisfied by the null object:

\[
(3) \quad \text{is-}(x) =
\]

\[
(x = \emptyset)
\]

The following special predicates and functions are used:

\[
(4) \quad \text{is-id} =
\]

Note: This predicate characterizes an infinite class of elementary objects which are different from all elementary objects (denoted by names written in capital letters), <>, *, [ ], all values, value representations and unique names.
(3) \( \text{is-n} = \)
Note: This predicate characterizes the enumerably infinite set of unique names \( n_0, n_1, n_2, \ldots \) which are all elementary objects.

(5.1) \( \text{is-ident} = \)

Note: This predicate characterizes the enumerably infinite set of attention identifications.

(6) \( \text{is-ids-n} = \)

Note: This predicate characterizes the enumerably infinite set of data set names.

(7) \( \text{is-set} = \)

\( \text{is-ob-set} \)

Note: The meaning of this predicate is explicated in 1.2.3.3.

The above mentioned special functions are now formalized:

(8) \( \text{sel(idl)} = \)

(9) \( \text{is-id-list(idl-1)} \land \text{is-id-list(idl-2)} \land \text{idl-1} \neq \text{idl-2} \Rightarrow \text{sel(idl-1)} \neq \text{sel(idl-2)} \)

(10) \( s(i) = \)

(11) \( \text{is-intg-val(i)} \land \text{is-intg-val(j)} \land i \neq j \Rightarrow s(i) \neq s(j) \)

Ref.: \( \text{is-intg-val 9-3(5)} \)

(12) \( \text{elem(i)} = \)

Note: This function maps integer values into unique names and is defined by:

(13) \( \text{is-intg-val(i)} \Rightarrow \text{is-n\#elem(i)} \)

Ref.: \( \text{is-intg-val 9-3(5)} \)

(14) \( \text{is-intg-val(i)} \land \text{is-intg-val(j)} \land i \neq j \Rightarrow \text{elem(i)} \neq \text{elem(j)} \)

Ref.: \( \text{is-intg-val 9-3(5)} \)

(15) \( \text{mk-id(cl)} = \)

Note: This function maps lists of character values into identifiers \( \text{is-id} \) and is defined by:

1. NOTATION AND CONVENTIONS 7
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

1.1.4 LISTS

Lists form a special class of objects. A number of functions, operations, and abbreviations is introduced for the manipulation of lists, which correspond closely to the conventional means for that purpose.

The predicate is-list holds for any list, it is defined as

(18) is-list(list) =
    is-<>(list) ∨ (∃x-1,...,x-n) list = µ0(<elem(1):x-1>,<elem(2):x-2>, ..., <elem(n):x-n>) & ~is-<>(x-2) & ... & ~is-<>(x-n))

An abbreviation for denoting elements of a list is introduced.

elem(i,list) = elem(i)(list) for is-list(list).

(20) is-list-1(list) =
    is-list(list) & ~is-<>(list)

Note: This predicate excludes especially for n=0 is-<>(list)

The length of a list is defined as the largest index of an element which is not the null object.

(21) length(list) =
    is-<>(list) → 0
    is-list(list) → (∃i)(~is-<>•elem(i,list) & is-<>•elem(i + 1,list))
T → error

The following functions yield, when applied to a list, the head (which is the first element of a list if it exists), the last element of a list (if it exists), the tail of a list (which is the original list except the first element) and the first (which is the original list except the last element).
Whenever a statement list, if-statement or access statement is to be interpreted, the complete item is inserted into the text component of CI and the index component of CI is set accordingly. When a statement, other than the last one of a statement list, has been interpreted the index is incremented by 1 and the next statement is interpreted.

Whenever during the interpretation of a statement at a statement list, or a component of an if-statement or access statement is to be interpreted, by entering the text and index into the control information the old text and index components of CI would be overwritten. But since they are needed for continuation after completion of the statement, they have to be reserved somewhere during the interpretation of the nested statement list, if-statement, or access statement. For this purpose the complete control information CI is stacked into a third component s-ci(CI) of the new control information whenever a new text component is inserted, i.e., whenever a new level of statement nesting is entered.

Similarly, the control CI is stacked in a fourth component of the control information, whenever a new level of statement nesting is entered. This control contains the actions to be performed after return from the nested statement level, while the control CI itself specifies only the actions to be performed for interpretation of the new level.

The complete control information thus works as a stack localizing uniquely that statement within the nested structure of statement lists, if-statements and access statements of the block activation currently being interpreted. Whenever a new nested level is entered a new level is introduced, whenever a level is completed the old control information and control is reinstalled from CI. This mechanism is started by initializing CI to 0 on start of a block activation.

The control information CI is a block local state component stacked into the dump Q by nested block activations and reinstalled by their epilogues. So, on return from a nested block activation, the flow of control continues at the point of interruption.

(17) \[
\text{is-ci} = \\
\text{is-g} \lor \\
\text{is-text:is-prop-st \ v \ is-g}, \\
\text{is-index:is-index \ v \ is-g}, \\
\text{is-ci:is-ci}, \\
\text{is-cis-c}, \\
\text{is-fol: (is-init:is-format-list \ v \ is-g),} \\
\text{is-expand:is-format-1-list \ v \ is-g})
\]

Note: In erroneous cases during a goto statement the text component may be any proper statement. Also during a goto statement, the index component may be 0. The s-fol component is used during edit directed get or put statements to control the expansion of the format specification.

(18) \[
\text{is-format-1} = \\
\text{is-format} \lor \text{is-13}
\]

1.1 ASSOCIATION OF IDENTIFIERS WITH MEANING

For each declaration occurring in the text of a block, a unique name n is created when the block is activated. This unique name n is copied into the program text of the block whenever the declared identifier (and the declared identifier lists in the case of a structure declaration) occurs, serving as a unique qualification of the identifier. Also entries are made, under the unique name n, into two state components, the attribute directory NA and the denotation directory DN, which hold information about the meaning of the declared identifier.
Thus it is possible, when interpreting a piece of program text, to have access to all information necessary to interpret occurring identifiers correctly.

3.3.1 CREATION OF UNIQUE NAMES

In many instances during interpretation of a PL/I program, so-called "unique names" are needed (e.g., for unique qualification of identifiers as described above). The only property required for them is that they are simple selectors which are different from each other. I.e., whenever a unique name n is needed, it must be guaranteed, that it is different from all unique names occurring elsewhere during the interpretation, except of course all occurrences of an explicit copy of n itself.

The function elem (which gives the different selectors for the components of lists) maps the integer values one-to-one into simple selectors. By applying elem to a state component, the unique name counter UN whose only purpose is to record which values have already been used, a new simple selector is found. The unique name counter is 0 in the initial state and is incremented by 1 on creation of a unique name (terminology used throughout this document). The unique names satisfy the predicate is-n.

Note: The property, that the unique names are ordered by their correspondence to the integers, is never used during the interpretation, except to guarantee their distinctness. The uses of the values of elem as list selectors and as unique names are never in conflict with each other.

(19) is-un =
    is-intg-val

ref.: is-intg-val 9-3(5)

(20) un-name =
    PASS: elem(UN)
    s-un: UN + 1

3.3.2 THE ATTRIBUTE DIRECTORY AT

The attribute directory AT contains all declarations of all block activations, which have been established, entered under the unique names associated with the declaration by the block activations. It is the text of the declarations as occurring in the program and modified by the prepass (cf. 4.3) and by the unique qualification of names by the current and all (statically) surrounding block activations.

The entries into the attribute directory AT, which is a global state component, are made by the prologue of each block activation. They are never changed or removed. After a block activation has terminated, generally its entries, though still contained in the attribute directory, are no longer accessible since no program text qualified by the corresponding block activations is used any more. The only exceptions are entry, label and format constants, which could be displaced into other block activations by entry or label variables; their misuse is tested by means of the block activation name.
(21) \text{is-at} =
\left(\left\{ \text{<is-decl> | | is-n(n) v n = s-builtin} \right\} \right)

Note: To guarantee correct interpretation of builtin function identifiers during
the prepass, i.e. before their declarations have been entered by any block
activations, their declaration BUILTIN is contained under the fixed
selector s-builtin in the attribute directory AT of the initial state.
This selector s-builtin is used to qualify all builtin identifiers in the
program text before the prepass.

3.3.3 THE DENOTATION DIRECTORY DN

The denotation directory DN contains the denotations of the declarations of all
block activations, which have been established, entered under the unique names
associated with the declarations by the block activations. These entries into the
denotation directory DN, which is a global state component, are made by the
prologue of each block activation. They are never changed or removed. For
accessing an entry after termination of its block activation, the same applies as
for the attribute directory AT above.

The denotation of a declaration consists of all information necessary for
correct interpretation of the declared identifier as far as it is fixed throughout
the block activation and not yet contained in the text of the declaration itself.
The information contained in the text of the declaration is to be found in the
attribute directory AT. The information which changes during the block activation
(e.g. values of variables) is to be found in other state components by an
information chain via a unique name in the denotation directory.

The denotations of the different types of declarations are the following:

1) For a proper variable (or a parameter) the denotation is a unique name b,
called its "aggregate name", used as selector to the aggregate directory AG
to find the generations and by that the allocated storage and finally the
values of the variable.

2) For a defined variable the denotation is its evaluated aggregate attribute.

3) For an entry constant the denotation consists of four components: The
   corresponding procedure body, the declared entry identifier (to be used to
   find the right entry point into the body when called), the block activation
   name (to be used for testing) and the block prefix part of GP (to be
   inherited into the procedure when called). The block activation name and
   the block prefix part are taken from that block activation in which the
   denotation was established. This is the outermost block activation of the
   program (i.e. both components are 0) for an external entry constant, and
   the block activation in which it was declared for an internal entry
   constant.

4) For a file constant the denotation is a unique name f, called its "file
   name", used as selector to the file directory FD to find all information
   about the corresponding file, finally its data set in the external storage.

5) For a label constant the denotation consists of the block activation name
   of the block activation in which it was declared (i.e. into which a goto
   with that label shall lead) and of an index list specifying a statement
   within that block activation.

6) For a format constant the denotation consists of the declared format list
   (to be used by means of remote formats in get and put statements) and three
   components used for testing only, namely: the block activation name, an
   identifier uniquely identifying the format constant within the block
   activation, and a condition prefix part.
7) For an attention identifier the denotation is a unique name used as selector into the attention environment directory to find its evaluated environment attribute.

8) Based, generic, builtin and programmer-named condition declarations need no denotations. The knowledge of their declarations in the attribute directory is sufficient for their interpretation.

Apart from these denotations of declarations, generic builtin functions, which are assigned to entry variables (on parameter passing), receive denotations entered under newly created unique names immediately before assignment (cf. 12.4). Such a denotation consists of the builtin identifier and the list of the data attributes of the parameters allowed on a call. This list of data attributes characterizes a single member out of the generic family denoted by the builtin identifier.

\[
\text{is-den} = (\text{is-n} \lor \text{is-eva} \lor \text{is-entry-den} \lor \text{is-label-den} \lor \text{is-format-den} \lor \text{is-builtin-den})
\]

\[
\text{is-entry-den} = \\
\langle\text{s-body:is-body}, \text{s-id:is-id}, \text{s-bs:is-bs}, \text{s-bpp:is-pref-part}\rangle
\]

Ref.: is-ba 3-7(15)
\text{is-pref-part} 3-21(79.1)

\[
\text{is-label-den} = \\
\langle\text{s-ba:is-name}, \text{s-sl:is-index-list}\rangle
\]

\[
\text{is-format-den} = \\
\langle\text{s-format-list:is-format-list}, \text{s-ba:is-name}, \text{s-ident:is-id}, \text{s-spp:is-pref-part}\rangle
\]

Ref.: is-pref-part 3-21(79.1)

\[
\text{is-builtin-den} = \\
\langle\text{s-id:is-id}, \text{s-da-list:is-da-list}\rangle
\]

3.3.4 THE AGGREGATE DIRECTORY \(\mathbb{A}_G\)

The aggregate directory \(\mathbb{A}_G\) contains for each proper variable and for each dummy variable the list of its generations, entered under a unique name \(b\). This unique name \(b\), called the "aggregate name" of the variable is for a proper variable to be
found as its denotation in the denotation directory DN.

A generation gen contains all information necessary to access the storage § and
to determine the values of a variable. It consists of three components:

1) The pointer part s-pp(gen) specifies in which parts of the storage § the
variable is allocated. For a newly allocated variable it is always a
single pointer. The storage associated with variables that are
cross-sections of arrays or sub-structures of arrays of structures may be
disconnected. In this case the pointer part is a (possibly nested) list of
pointers.

2) The mapping information s-mi(gen) provides the argument to the storage
mapping function determining how the various parts of the variable are
located in those parts of storage which are given by the pointer part.

3) The aggregate attribute s-eva(gen) is relevant for referencing the
variable. It specifies the aggregate structure and the data attributes of
the scalar components. The scalar data attribute of a scalar component is
the key for the function which determines the actual value of the value
representation found in the storage by means of the pointer part and the
mapping information.

Whenever a variable is allocated a generation for it is entered into the
aggregate directory AG, whenever a variable is freed its current generation is
removed. This is done for the variables of the different storage classes in
different way:

For an automatic variable the list of generations consists of a single
allocation by the prelogue of the block activation in which it
is declared and freed by the corresponding block epilogue. The information which
variables are to be freed contained in the epilogue information.

A controlled variable the prepass of the program initializes the generation
list by making an entry which gives the aggregate structure of the variable (this
is used for testing if a controlled variable which has not yet any generation is
passed to a controlled parameter). Each allocate statement for the variable
enters a generation on top of the list, each free statement removes the top
generation. Always the top generation is taken as the current generation of the
variable. Controlled variables are the only ones which may have more than one
generation in the generation list.

For a static variable the prepass allocates a generation and enters it together
with an * as a two-element list into the aggregate directory. It is freed at
program end.

Dummy variables are allocated and entered into the aggregate directory in the
same way as static variables when they are needed. Usually they are freed when
they are no longer necessary.

The aggregate directory is a task local state component, i.e., each task
maintains its own one. When a new task is created, it inherits only the top
entries of the aggregate directory of the attaching task. When the task is
terminated, all generations, except the bottom ones are freed. This applies for
those generations of controlled variables which have been allocated by the task,
for dummy variables created by a task, and for the static variables in the case of
the main task. Also freeing of controlled variables by means of a programmed free
statement is possible only for all but the bottom generations. By this mechanism
it is guaranteed that each task frees those and only those generations which were
allocated by the task itself. The only exception for this rule are dummy
arguments passed to the task (cf. 5.1).

When generations of variables are passed to non controlled parameters (cf.
6.2.3.2), they are entered under the aggregate name of the parameter as
one-element lists in order to avoid that they are freed by the aggregate name of
the parameter.

3. STATE COMPONENTS AND COMPUTATION OF THE PL/I MACHINE 13
The allocation and freeing of base storage is not controlled by the aggregate directory \texttt{AG}, but by the \texttt{s-free-set} component of the task-event specification \texttt{TE}. The allocation and freeing of buffers for record input and output statement is controlled by the file union directory \texttt{FU}.

(28) \texttt{is-ag} =
\begin{align*}
&\begin{cases}
\text{is-ag-entry-list} \mid \text{is-n(b)}
\end{cases}
\end{align*}

(29) \texttt{is-ag-entry} =
\begin{align*}
\text{is-gen} \land \text{is-}^* \land
&\begin{cases}
\text{s-stg-cl:is-CTL}, \\
\text{s-eva:is-eva}
\end{cases}
\end{align*}

Note: Aggregate directory entries which are no generations occur only as bottom entries for controlled, static and dummy variables.

(30) \texttt{is-gen} =
\begin{align*}
&\begin{cases}
\text{s-eva:is-eva}, \\
\text{s-mi:is-eva}, \\
\text{s-pp:is-pp}
\end{cases}
\end{align*}

Note: The \texttt{s-eva} and \texttt{s-mi} components have the same aggregate structure, they may differ only in array bounds and string lengths, if the generation is produced by correspondence defining.

(31) \texttt{is-pp} =
\begin{align*}
\text{is-}^\ast \text{-val} \mid \text{is-pp-list}
\end{align*}

Ref.: \texttt{is-pp-val} 3-15(36)

\section{3.1 THE STORAGE PART \texttt{S}}

The storage part \texttt{S} of the PL/I machine is a model of actual machine storage. The model shows the essential properties of machine storage without, however, showing the features of a specific machine.

The storage part is defined implicitly by stating properties of and relations between the various elementary storage functions. This means that storage is not described as an object. There are many possible realizations as an object, but it is inessential as to which one is chosen. The definition leaves these possibilities open.

Storage is used to represent values of some kind. Storage parts, therefore are called value representations. Also the entire storage part \texttt{S} in its nature is a value representation, and satisfies the predicate \texttt{is-vr}.

\textbf{Metavariabes}
\begin{itemize}
\item \texttt{p} \hspace{1cm} \texttt{is-pp-val} \hspace{1cm} pointer value
\end{itemize}
3.4.1 VALUE REPRESENTATIONS AND POINTER VALUES

A value representation \( \text{vr} \) satisfies the predicate

\[
(32) \quad \text{is-vr}(\text{vr}) = 1
\]

For any value representation the function

\[
(33) \quad \text{size}(\text{vr}) = \]

is defined. The range of the function is characterized by the predicate

\[
(34) \quad \text{is-size}(z) = \]

which is implementation defined:

\[
(35) \quad \text{is-size} \cdot \text{size}(\text{vr})
\]

Parts of value representations may be selected by functions \( p \) which are called pointer values, and which satisfy the predicate

\[
(36) \quad \text{is-ptr-val} = \]

A pointer value may or may not be applicable to a certain value representation. The predicate

\[
(37) \quad \text{is-applic}(p,z) = \]

is true if \( p \) is applicable to a value representation of size \( z \), false otherwise. The result of the application is a value representation. The size of the resulting value representation is determined by the function

\[
(38) \quad \text{size-1}(p) = \]

\[
(39) \quad \text{is-applic}(p,\text{size}(\text{vr})) \supset \text{is-vr}\cdot p(\text{vr}) \& \text{size}\cdot p(\text{vr}) = \text{size-1}(p)
\]

Provided applicability, pointer values may be composed:

\[
(40) \quad \text{is-applic}(p-1,\text{size-1}(p-2)) \supset \text{is-ptr-val}(p-1\cdot p-2)
\]

\[
(41) \quad \text{is-applic}(p-1,\text{size-1}(p-2)) \& \text{is-applic}(p-2,z) \supset \text{is-applic}(p-1\cdot p-2,z)
\]
Two pointer values may select overlapping parts of value representations. If the selected parts do not overlap, the predicate

\[ \text{is-indep}(p_1, p_2) = \]

is true, false otherwise. The independence relation is symmetric. Further, a part is never independent of the whole, and parts of independent parts are again independent. These properties are expressed by:

\[ \neg \text{is-indep}(p, p) \]

\[ \text{is-indep}(p_1, p_2) \implies \text{is-indep}(p_2, p_1) \]

\[ \text{is-ptr-val}(p_1 * p_2) \implies \neg \text{is-indep}(p_1 * p_2, p_2) \]

\[ \text{is-indep}(p_1, p_2) \land \text{is-ptr-val}(p * p_2) \implies \text{is-indep}(p_1, p * p_2) \]

3.4.2 ELEMENTARY ASSIGNMENT

A part of a value representation \( vr \), identified by a pointer value \( p \), can be replaced by the elementary assignment function:

\[ \text{el-ass}(vr_1, p, vr) = \]

The result is a value representation:

\[ \text{is-vr} * \text{el-ass}(vr_1, p, vr) \]

This differs from \( vr \) only in that its \( p \)-part is now \( vr_1 \). This is expressed by:

\[ \text{size} * \text{el-ass}(vr_1, p, vr) = \text{size}(vr) \]

\[ \text{size}(vr_1) = \text{size}(p) \implies p * \text{el-ass}(vr_1, p, vr) = vr_1 \]

\[ \text{is-indep}(p_1, p_2) \land \text{size}(vr_1) = \text{size}(p_2) \implies p_1 * \text{el-ass}(vr_1, p_2, vr) = p_1(vr) \]

with the assumption that \( \text{is-applic}(p, \text{size}(vr)) \), \( \text{is-applic}(p_1, \text{size}(vr)) \), and \( \text{is-applic}(p_2, \text{size}(vr)) \). Assignment may be super-imposed:

\[ \text{is-applic}(p_1, \text{size}(p_2)) \land \text{is-applic}(p_2, \text{size}(vr)) \land \text{size}(vr_1) = \text{size}(p_2) \implies \text{el-ass}(vr_1, p_1 * p_2, vr) = \text{el-ass}(\text{el-ass}(vr_1, p_1, p_2(vr)), p_2, vr) \]

3.4.3 ELEMENTARY ALLOCATION

Those parts of a value representation (the main storage or an area) which have been reserved by allocation of variables, are identified in the allocation state. The allocation state is a set of pointers identifying the reserved parts, it is obtained by the function
(53)  alloc-state(vr) =
applied to the value representation.
(54) \textit{is-ptr-val-set\textcdot alloc-state}(vr)

On allocation the pointer value identifying the part to be reserved is added to the allocation state by using the elementary allocation function.

(55) \textit{el-alloc}(p, vr, type) =

The result of the elementary allocation function is a value representation which differs from vr in that its allocation state is amended:

(55.1) \textit{is-vr\textcdot el-alloc}(p, vr, type)

The allocation is possible if the implementation defined predicate

(56) \textit{is-free-space}(p, vr, type) =

is true, where type indicates the type of the allocation. The predicate guarantees the applicability of p to vr, and the independence of the p-part of all other parts identified in the allocation state.

(57) \textit{is-free-space}(p, vr, type) = \textit{is-applic}(p, \textit{size}(vr)) & (p-1 \notin \textit{alloc-state}(vr) \Rightarrow \textit{is-indep}(p-1, p))

The properties of the allocation function are expressed by:

(58) \textit{size\textcdot el-alloc}(p, vr, type) = \textit{size}(vr)

(59) \textit{is-free-space}(p, vr, type) \Rightarrow \textit{alloc-state\textcdot el-alloc}(p, vr, type) = (\textit{alloc-state}(vr) \cup \{p\})

(60) \textit{is-free-space}(p, vr, type) \& p-1 \notin \textit{alloc-state}(vr) \Rightarrow p-1\textcdot \textit{el-alloc}(p, vr, type) = p-1(vr)

(61) p \notin \textit{alloc-state}(vr) \& \textit{size}(vr-1) = \textit{size-1}(p) \Rightarrow \textit{alloc-state\textcdot el-ass}(vr-1, p, vr) = \textit{alloc-state}(vr)

3.4.9 ELEMENTARY FREEING

The allocation state can be reduced by using the elementary freeing function

(62) \textit{el-free}(ps, vr) =

The result is a value representation:

(62.1) \textit{is-vr\textcdot el-free}(ps, vr)

This differs from vr in that its allocation state is reduced by the set of pointer values ps. The properties of the elementary freeing function are expressed by:
3.5 STATE COMPONENTS FOR ATTENTIONS AND CONDITIONS

The attention directory \( \mathcal{A} \), the attention enabling state \( \mathcal{E} \), and the attention environment directory \( \mathcal{S} \) are solely used to describe the enabling and disabling mechanism of attentions in the various tasks and to stack the attention information.

The condition status \( \mathcal{C} \) contains the enabling information for prefix controlled conditions. The other two parts of \( \mathcal{C} \), holding information for interpreting on-units and condition built-in functions, are used for all conditions.

3.5.1 ATTENTION DIRECTORY \( \mathcal{A} \)

This global directory holds the information relevant for executing access statements and asynchronous interrupts. The updating of this directory is done by enable and disable statements and task termination.

The attention directory \( \mathcal{A} \) contains for each attention identification (a selector satisfying the implementation defined predicate \( \text{is-ident} \)) four components: the first contains the stack of attention information, which is updated from outside the PL/I machine; the second specifies the task for which the attention is enabled; the third characterizes the enabling mode; the last collects the names of the tasks which have this attention associated.

\[
(66) \text{is-an} = \\
\{ [\text{ident:is-attention} \mid \text{is-ident(ident)}] \}
\]

\[
(67) \text{is-attention} = \\
(\text{s-info:is-info-list}, \\
\text{<task:is-tn>}, \\
\text{<spec:is-ACC v is-ASYN v is-ACC-1>}, \\
\text{s-assoc:is-tn-set})
\]

Ref.: is-tn 3-4(6)

Note: Whenever an asynchronous interrupt is interpreted the enabling mode is changed to ACC-1, to differentiate from normal ACC.

\[
(68) \text{is-attn-occ} = \\
(\text{<ident:is-ident>}, \\
\text{s-info:is-info})
\]
(70) \[\text{is-info} = (\langle\text{s-struct:is-eva}\rangle, \langle\text{s-op:is-op}\rangle)\]

Ref.: \text{is-op 9-9(34)}
3.5.2 ATTENTION ENABLING STATE EN

This task-local state component holds the enabling information of all attentions in a task. One component lists the attentions enabled in the task, another one the attentions associated with the task. The component s-wait-list enumerates the attentions which are only associated with the task but are not specified with an event, so that the task must wait until all these attentions are enabled for the task. The last component is a set which specifies for each event name the corresponding attentions, which are associated with (but not enabled in) the task.

(71) is-en =

\[
(\langle\text{s-enab-list:is-eatttn-cond-list}\rangle, \\
\langle\text{s-assoc-list:is-eatttn-cond-list}\rangle, \\
\langle\text{s-wait-list:is-eatttn-cond-list}\rangle, \\
[\langle\text{tn:is-eatttn-cond-list}\rangle | | \text{is-tn}(\text{tn})])
\]

Ref.: is-tn 3-4(6)

(72) is-eatttn-cond =

\[
(\langle\text{s-ident:is-ident}\rangle, \\
\langle\text{s-spec:is-ACC} \vee \text{is-ASYN} \vee \text{is-0}\rangle, \\
\langle\text{s-tn:is-tn-set} \vee \text{is-0}\rangle)
\]

Ref.: is-ident 1-7(5.1)

3.5.3 ATTENTION ENVIRONMENT DIRECTORY EV

This attention directory contains for each unique name an evaluated environment which is used to create the attention identification.

(73) is-ev =

\[
(\langle\text{b:is-ea} \mid \text{is-n}(\text{b})\rangle)
\]

Ref.: is-ea 11-22(58)

3.5.4 CONDITION SELECTORS

For the access to components of major subparts of GS condition selectors are generated by the specific function cond-sel which returns a selector value. The auxiliary function sel is used which generates a selector value from a name list. The auxiliary function cond-io-sel generates selectors from i/o-condition names.

The condition is an evaluated condition which satisfies the predicate is-COND. The evaluated condition differs from the condition described by the abstract syntax only in three points: the check condition which is a reference now; the i/o condition; the attention condition which mainly consists of the attention identification.

3. STATE COMPONENTS AND COMPUTATION OF THE PL/I MACHINE 19
18:1 LAB VIENNA

ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

31 October 1969

(73.1) cond-sel(cond,at) =

  \[\text{is-CONV}(\text{cond}) \rightarrow \text{s-conv}\]

  \[\text{is-FOFL}(\text{cond}) \rightarrow \text{s-fofl}\]

  \[\text{is-GFL}(\text{cond}) \rightarrow \text{s-gfl}\]

  \[\text{is-SIZE}(\text{cond}) \rightarrow \text{s-size}\]

  \[\text{is-STR$(\text{cond}) \rightarrow \text{s-str}\}

  \[\text{is-STR$}(\text{cond}) \rightarrow \text{s-strz}\]

  \[\text{is-UFL}(\text{cond}) \rightarrow \text{s-ufl}\]

  \[\text{is-ZDIV}(\text{cond}) \rightarrow \text{s-zdiv}\]

  \[\text{is-AREA}(\text{cond}) \rightarrow \text{s-area}\]

  \[\text{is-ERROR}(\text{cond}) \rightarrow \text{s-error}\]

  \[\text{is-FINISH}(\text{cond}) \rightarrow \text{s-finish}\]

  \[\text{is-check}(\text{cond}) \& \text{is-EXT$\cdot$s-scope}(\text{s-n}(\text{cond}) (\text{at})) \rightarrow (\text{sel} \cdot \text{s-id-list}(\text{cond})) \cdot \text{s-check}\]

  \[\text{is-check}(\text{cond}) \rightarrow (\text{s-n}(\text{cond})) \cdot (\text{sel} \cdot \text{s-id-list}(\text{cond})) \cdot \text{s-check}\]

  \[\text{is-f-cond}(\text{cond}) \rightarrow (\text{s-f}(\text{cond})) \cdot (\text{cond} \cdot \text{io-sel}(\text{s-cond}(\text{cond})))\]

  \[\text{is-progr-named-cond}(\text{cond}) \rightarrow (\text{sel} (\text{<s-id}(\text{cond})) \cdot \text{s-progr-named-cond}\]

\[\text{is-attn-cond}(\text{cond}) \rightarrow (\text{s-ident}(\text{cond})) \cdot \text{s-attn}\]

\text{for: is-econd}(\text{cond}), \text{is-at}(\text{at})\

Ref.: \text{is-at} 3-11(21)

(75) cond-io-sel(iocond) =

  \[\text{is-BOV}(\text{iocond}) \rightarrow \text{s-bov}\]

  \[\text{is-EOV}(\text{iocond}) \rightarrow \text{s-eov}\]

  \[\text{is-ENDP}(\text{iocond}) \rightarrow \text{s-endf}\]

  \[\text{is-ENDP}(\text{iocond}) \rightarrow \text{s-endp}\]

  \[\text{is-KEY}(\text{iocond}) \rightarrow \text{s-key}\]

  \[\text{is-NAM$E}(\text{iocond}) \rightarrow \text{s-name}\]

  \[\text{is-REC}(\text{iocond}) \rightarrow \text{s-rec}\]

  \[\text{is-TMT}(\text{iocond}) \rightarrow \text{s-tmt}\]

  \[\text{is-UNDF}(\text{iocond}) \rightarrow \text{s-undf}\]

  \[\text{is-PEND}(\text{iocond}) \rightarrow \text{s-pend}\]

\text{for: is-io-cond}(\text{iocond})

20 3. STATE COMPONENTS AND COMPUTATION OF THE PL/I MACHINE
(76) \[ \text{is-COND} = \]
\[ \text{is-COND} \land \text{-is-Check-COND} \land \text{-is-Attn-COND} \land \text{is-IO-COND} \lor \text{is-Check} \]
\[ \text{is-Attn-COND} \lor \text{is-IO-COND} \lor \text{is-f-COND} \]

(77) \[ \text{is-Check} = \]
\[ \text{is-ref} \]

(78) \[ \text{is-f-COND} = \]
\[ (<\text{is-f-cond:is-st-prt} \lor \text{is-n}>, \]
\[ <\text{is-cond:is-IO-COND}>, \]
\[ <\text{is-cbif:is-cbif}>) \]

Ref.: is-s-st-prt 3-24(90)

3.5.5 THE CONDITION STATE Cs

The condition state Cs is composed of four major subparts. The block-prefix-part selected by s-bpp contains the enabling information for all conditions which may be prefixed to a block. The statement-prefix-part selected by s-spp contains all prefixes valid during interpretation of a statement.

The condition-action-part selected by s-cap contains the actions which have to be performed if an enabled condition is raised and interpreted.

The condition-built-in-function-part contains components for specifying the values returned on references to these functions and some auxiliary parts for obtaining these values.

(79) \[ \text{is-cs} = \]
\[ (<\text{s-bpp:is-pref-part}>, \]
\[ <\text{s-spp:is-pref-part}>, \]
\[ <\text{s-cap:is-cond-act-part}>, \]
\[ <\text{is-cbif:is-cond-bif-part}>) \]

(79.1) \[ \text{is-pref-part} = \]
\[ (<\text{cond-sel(Cond,at):is-opt}>) \]
\[ (<\text{is-prefix-cond(Cond)} \land \text{-is-check-cond(Cond)} \lor \text{is-check(Cond)} \land \text{is-at}(at))> \]

Ref.: is-at 3-11(21)

(81) \[ \text{is-cond-act-part} = \]
\[ (<\text{cond-sel(Cond,at):is-cond-act}>) \]
\[ (<\text{is-COND(Cond)} \land \text{is-at}(at))> \]

Ref.: is-at 3-11(21)
(52) \( \text{is-cond-act} = \)

\[
(<\text{s-cond:is-ec}\text{on3}>,
 <\text{s-bpp:is-pref-part}>,
 <\text{s-snap:is-opt}>,
 <\text{s-on-unit:is-st v is-SYSTEM}>)
\]

(82.1) \( \text{is-cond-bif-part} = \)

\[
(<\text{s-onloc:is-id v is-0}>,
 <\text{s-oncode:is-intg-val v is-0}>,
 <\text{s-oncount:is-intg-val v is-0}>,
 <\text{s-onfile:is-id v is-0}>,
 <\text{s-onfile-def:is-id v is-0}>,
 <\text{s-onkey:is-char-val-list v is-0}>,
 <\text{s-datafield:is-char-val-list v is-0}>,
 <\text{s-onchar:is-intg-val v is-0}>,
 <\text{s-onsource:is-gen v is-ps-gen v is-0}>,
 <\text{s-onident:is-gen},
 <\text{s-input:is-T v is-F}>) v is-0>,
 <\text{s-onattn:is-ptr-val v is-0}>,
 <\text{s-entry:is-id v is-0}>,
 <\text{s-type:is-SIGNAL v is-0}>,
 <\text{s-abn-ret:is-abn-ret},
 <\text{s-cond:is-eattn-cond} v is-0>)
\]

Ref.: is-intg-val 9-3(5)
is-char-val 9-3(6)
is-gen 3-14(30)
is-ps-gen 12-67(169)
is-ptr-val 3-15(36)

(84) \( \text{is-abn-ret} = \)

\[
(<\text{s-en:is-n}>) v
(<\text{s-u:is-n}>,
 <\text{s-cond:is-BOV v is-BOV}>) v is-0
\]

(85) \( \text{is-cbif} = \)

\[
(<\text{s-oncode:is-intg-val v is-0}>,
 <\text{s-oncount:is-intg-val v is-0}>,
 <\text{s-onfile:is-id v is-0}>,
 <\text{s-onkey:is-char-val-list v is-0}>,
 <\text{s-datafield:is-char-val-list v is-0}>,
 <\text{s-onchar:is-intg-val v is-0}>,
 <\text{s-onsource:is-gen v is-ps-gen v is-0}>,
 <\text{s-onident:is-gen},
 <\text{s-input:is-T v is-F}>) v is-0>,
 <\text{s-onattn:is-ptr-val v is-0}>,
 <\text{s-type:is-SIGNAL v is-0}>,
 <\text{s-abn-ret:is-abn-ret},
 <\text{s-cond:is-eattn-cond} v is-0>)
\]

Ref.: is-intg-val 9-3(5)
is-char-val 9-3(6)
is-gen 3-14(30)
is-ps-gen 12-67(169)
is-ptr-val 3-15(36)
3.7.2 THE ENVIRONMENT STEP

The predicate is-env-step(t-1,t-2) tests whether t-2 is an admissible modification of t-1. Modifications by the environment may concern the internal storage, the external storage, the time part, the message part, and the attention directory.

\[
\text{is-env-step}(t-1,t-2) = \mu(t-1; s-s: s: s: s(t-2)), \\
\text{is-time-step}(t-1, t-2), \\
\text{is-reply-step}(s: s: s: m: s: m(t-2)), \\
\text{is-an-step}(s: a: s: (a-l), (a-2)) \\
\text{&} \\
\text{is-step}(s: s(t-1), s: s(t-2)) \\
\text{&} \\
\text{&} \\
\text{is-time-step}(s: t: s: t: d(t-1), s: t: s: t: d(t-2)) \\
\text{&} \\
\text{is-reply-step}(s: r: s: m(t-1), s: r: s: m(t-2)) \\
\text{&} \\
\text{is-an-step}(s: a: n(t-1), s: a: n(t-2))
\]

(115) is-s-step(s-1, s-2) =
\[
is-vr(s-2) \& \text{size}(s-1) = \text{size}(s-2) \& \text{alloc-state}(s-1) = \text{alloc-state}(s-2) \&
\{ \forall p (p \in \text{alloc-state}(s-1) \Rightarrow p(s-1) = p(s-2)) \}
\]

Ref.: is-vr 3-15(32)
size 3-15(33)
alloc-state 3-16.1(53)

Note: The parts of the internal storage which are identified by the allocation state must not be modified by the environment.

(117) is-ds-dir-step(ds-dir-1, ds-dir-2) =
\[
is-ds-dir(ds-dir-2) \& \\
\{ \forall ds-n \} (\text{is-}0*ds-n(ds-dir-1) \Rightarrow \text{is-}0*ds-n(ds-dir-2) \& \\
\text{is-time-}ds-n(ds-dir-1) \Rightarrow \text{is-time-}ds-n(ds-dir-2))
\]

Ref.: is-ds-dir 3-27(101)

Note: Data sets and transmission error flags must not disappear. However, existing data set members may be replaced and/or additional members may be entered. A setting of the transmission flag to INT will be interpreted as a transmission error through the next following data transmission to or from the flagged data set.

(118) is-time-step(time-1, time-2) =
\[
is-intg-val(time-2) \& \text{time-2} \geq \text{time-1}
\]

Ref.: is-intg-val 9-3(5)

Note: The value of the time component is required not to decrease in the course of the computation.
(119) \[ \text{is-reply-step}(\text{reply-1}, \text{reply-2}) = \]
\[ \text{is-named-message-list}(\text{reply-2}) \& \]
\[ (\text{reply})(\text{is-list}(\text{reply}) \& \text{reply-2} = \text{reply-1} \text{-reply}) \]

Ref.: is-named-message 3-28(108)

Note: Reply messages may be received from the environment. They are interpreted by the display statement (cf. 11.8).

(120) \[ \text{is-an-step}(\text{an-1}, \text{an-2}) = \]
\[ \text{is-an}(\text{an-2}) \& (\text{Vident})(\text{is-0} \text{-ident}(\text{an-1}) \Rightarrow (\text{info})(\text{ident}(\text{an-2}) = \text{attn}_1)) \]

where:
\[ \text{attn}_1 = \mu(\text{ident}(\text{an-1}) \text{-info:ident}(\text{an-1}) \text{-info}) \]

Ref.: is-an 3-18(66)

Note: Attentions may be received from the environment (cf. 10.3).

3.7.3 THE INTERRUPT STEP

In the interrupt step, tasks may be shifted into the active state (cf. 5.3.1), and the interruption of the normal execution in tasks which received an asynchronous attention is prepared (cf. 10.3.1). The predicate test-activate determines whether the activation of tasks is required.

(121) \[ \text{prep-interrupt}(\xi-1, \xi-2) = \]
\[ \text{prep-activate}(\xi-1, \text{prep-attn}(\xi-1, \xi-2)) \]

(122) \[ \text{prep-attn}(\xi-1, \xi-2) = \]
\[ \mu(\xi-2;\{<\text{tn}\text{-s-pa:prep-attn-1}(\text{tn}\text{-s-pa}(\xi_2),\mu_0(\text{ident}\text{-ident}))> | \text{tn} = \text{s-task*ident*s-an}(\xi-2) \& \text{info*ident*s-an}(\xi-1) \# \text{info*ident*s-an}(\xi-2) \}) \]

Ref.: prep-attn-1 10-14(39)

(123) \[ \text{prep-activate}(\xi-1, \xi-2) = \]
\[ \text{test-activate}(\xi-1, \xi-2) \rightarrow \]
\[ \mu(\xi-2;<\text{pa:activate-tasks*pa}(\xi-2)>,<\text{delay*td:delay}_1>) \]
\[ T \rightarrow \xi-2 \]

where:
\[ \text{delay}_1 = \{\text{time} | \text{time} \in \text{delay*td}(\xi-2) \& \text{time} > \text{time*td}(\xi-2) \} \]

Ref.: activate-tasks 5-12(42)
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

(3) \text{is-ready-for-alloc}(t,i,\text{at},\text{dn},\text{ag}) = \\
\quad ((30') (\text{is-\text{*\text{-}\text{a\text{-}\text{g}}\text{r\text{-}\text{e}\text{l}\text{e}\text{m}(i,t)}}) \Rightarrow \text{is-gen}(\text{gen}_1)) \& \\
\quad (\text{Vref}) (\text{is-alloc-dep}(\text{elem}(i,t),\text{ref},\text{at},\text{dn},\text{ag}) \& \text{is-CTL\text{-}\text{st\text{-}\text{g\text{-}\text{c\text{-}\text{l}(attr-ref}_n)) \& \\
\quad (3j) (\text{s-n}(\text{ref}) = \text{s-n}\text{-elem}(j,t)) \Rightarrow \text{is-gen}\text{-}\text{head}(\text{s-n}(\text{ref})(\text{dn})(\text{ag})))}

where:
\text{gen}_1 = \text{head}(\text{s-n\text{-elem}(i,t)(\text{dn})(\text{ag}))}
\text{attr-ref}_0 = \text{s-n}(\text{ref})(\text{at})

for: is-al-list(t)

Ref.: is-gen 3-14(30)

Note: An allocation is ready for execution if none of the expressions to be evaluated on its execution refers to an unallocated controlled variable specified in the same statement and, if the variable to be allocated is itself controlled and extents are to be determined by its most recent generation, the variable is already allocated.

(4) \text{is-alloc-dep}(t,\text{ref},\text{at},\text{dn},\text{ag}) = \\
\quad \text{is-dep}(\text{aggr}_0,\text{ref},\text{at}) \vee \neg \text{is-0}(\text{ptr}_0) \& \text{is-dep-2}(\text{ptr}_0,\text{ref},\text{at}) \vee \neg \text{is-0}(\text{area}_0) \& \\
\quad \text{is-dep-2}(\text{area}_0,\text{ref},\text{at})

where:
\text{aggr}_0 = \text{mix-spec}(\text{s-\text{aggr}(attr}_0),\text{s-\text{aggr}(t)},\text{s-\text{eva}(gen}_0))
\text{attr}_0 = \text{s-n}(t)(\text{at})
\text{gen}_0 = \text{head}(\text{s-n}(t)(\text{dn})(\text{ag}))
\text{ptr}_0 = (-\text{is-0}\&\text{s-\text{ptr}(t)} \Rightarrow \text{s-\text{ptr}(t)}, \text{T} \Rightarrow \text{s-\text{ptr}(attr}_0))
\text{area}_0 = (\neg \text{is-0}\&\text{s-\text{area}(t)} \Rightarrow \text{s-\text{area}(t)}, \text{T} \Rightarrow \text{area-expr(\text{ptr}_0,\text{at}))}

for: is-al(t)

Ref.: is-dep 6-6(16)
is-dep-2 6-6(18)
area-expr 6-28(81)

(5) \text{int-allocate}(t,i) = \\
\quad \text{S-C: int-next-st; int-allocate-list(remove(t,i)); int-allocate-l}(\text{elem}(i,t))

for: is-al-list(t)

Ref.: int-next-st 6-39(111)

Note: Once an allocation has been selected, it is interpreted by the instruction \text{int-allocate-l}, and the process of testing and selection is restarted with the list of remaining allocations.

7. ALLOCATION, INITIALISATION, AND FREEING OF VARIABLES 3
(6) $\text{remove}(\text{list}, i) =$
\begin{align*}
i &= 1 \rightarrow \text{tail}(\text{list}) \\
T &\rightarrow \langle \text{head}(\text{list}) \rangle ^* \text{remove}(\text{tail}(\text{list}), i - 1)
\end{align*}

(6.1) $\text{int-allocate-1}(t) =$
\begin{align*}
is-\text{CTL}\ast-\text{stg-cl}(\text{attr}_0) &\land is-\text{Q}\ast-\text{ptr}(t) &\land is-\text{Q}\ast-\text{area}(t) \rightarrow \\
\text{initialize}(\text{gen}, \text{aggr}_0); \\
\quad \text{gen}: \text{allocate-ctl}(n_0(\text{DN}), \text{eva}); \\
\quad \text{eva}: \text{eval-aggr}(\text{aggr}_1)
\end{align*}

\begin{align*}
is-\text{based}(\text{attr}_0) &\land is-\text{Q}\ast-\text{aggr}(t) &\land is-\text{Q}\ast-\text{area}(t) &\land is-\text{var-ref}(\text{ptr}_0, \text{AG}) &\land \\
is-\text{Q}\ast-\text{aggr-ref}(\text{ptr}_0, \text{AG}) \rightarrow \\
\text{initialize}(\text{gen}, \text{aggr}_0); \\
\quad \text{set-arg-ref}(\text{ptr}_0, \text{gen}, \text{aggr}_0); \\
\quad \text{gen}: \text{allocate-based}(\text{eva}); \\
\quad \text{eva}: \text{eval-alloc-aggr}(\text{aggr}_0)
\end{align*}

\begin{align*}
is-\text{based}(\text{attr}_0) &\land is-\text{Q}\ast-\text{aggr}(t) &\land is-var-ref(\text{ptr}_0, \text{AG}) &\land \\
is-var-ref(\text{area}_0, \text{AG}) \rightarrow \\
\text{initialize}(\text{gen}, \text{aggr}_0); \\
\quad \text{set-arg-ref}(\text{ptr}_0, \text{gen}, \text{aggr}_0); \\
\quad \text{gen}: \text{allocate-area}(\text{gen}, \text{eva}, \text{area}_0); \\
\quad \text{eva}: \text{eval-alloc-aggr}(\text{aggr}_0), \\
\quad \text{gen}: \text{eval-ref-gen}(\text{area}_0)
\end{align*}

$T \rightarrow \text{RECOV}$

where:
\begin{align*}
n_0 &= s-n(t) \\
\text{attr}_0 &= n_0(\text{AG}) \\
\text{aggr}_0 &= s-aggr(\text{attr}_0) \\
\text{aggr}_1 &= \text{mix-spec}(\text{aggr}_0, s-aggr(t), s-eva(\text{gen}_0)) \\
\text{gen}_0 &= \text{head}(n_0(\text{DN})(\text{AG})) \\
\text{ptr}_0 &= (\neg is-\text{Q}\ast-\text{ptr}(t) \rightarrow s-\text{ptr}(t), \\
&\quad T \rightarrow s-\text{ptr}(\text{attr}_0)) \\
\text{area}_0 &= (\neg is-\text{Q}\ast-\text{area}(t) \rightarrow s-\text{area}(t), \\
&\quad T \rightarrow \text{area-expr}(\text{ptr}_0, \text{AG}))
\end{align*}

for $\neg \text{is-al}(t)$

Ref.:
\begin{align*}
\text{initialize} &\quad 7-15(49) \\
\text{eval-aggr} &\quad 6-9(23) \\
\text{is-var-ref} &\quad 8-9(20.1) \\
\text{aggr-ref} &\quad 8-25(70.1) \\
\text{eval-ref-gen} &\quad 8-28(82) \\
\text{area-expr} &\quad 8-28(81)
\end{align*}

7.1.1 ALLOCATION OF CONTROLLED VARIABLES

Specification of array bounds, string lengths, and area sizes used for the allocation are mixed from those given in the allocate statement, in the declaration of the controlled variable, and in the most recent generation (if existent) of the controlled variable. Rules for overriding specifications are given by the function mix-spec.

4 7. ALLOCATION, INITIALIZATION, AND FREEING OF VARIABLES
(14.1) \( \text{mix-spec-scalar}(\text{aggr}, \text{al}, \text{eva}) = \)
\[
\text{is-al-string} \& \text{s-base} = \text{s-base}(\text{aggr}) \Rightarrow \\
\mu(\text{aggr}; \langle \text{s-length} \text{s-da} \text{(al)}, \text{s-length} \text{s-da} \text{(eva)} \rangle, \\
\langle \text{s-init} \text{mix-init} (\text{s-init} \text{(al)}, \text{s-init} \text{(aggr)}) \rangle) \\
\text{is-prop-area} \Rightarrow \\
\mu(\text{aggr}; \langle \text{s-size} \text{s-da} \text{(al)}, \text{s-size} \text{s-da} \text{(eva)} \rangle, \\
\langle \text{s-init} \text{mix-init} (\text{s-init} \text{(al)}, \text{s-init} \text{(aggr)}) \rangle) \\
\text{is-\#}(\text{al}) \Rightarrow \\
\mu(\text{aggr}; \langle \text{s-init} \text{mix-init} (\text{s-init} \text{(al)}, \text{s-init} \text{(aggr)}) \rangle) \\
T \Rightarrow \text{error}
\]
for: \( (\text{is-eva} \lor \text{is-aggr} \lor \text{is-\#}) \text{(eva)} \)

(16) \( \text{mix} \text{(spec-1, spec-2)} = \)
\[
\text{is-\#}(\text{spec-1}) \& \text{is-intg-val} \text{(spec-2)} \Rightarrow \text{bintg-const} \text{(spec-2)} \\
\neg \text{is-\#}(\text{spec-1}) \Rightarrow \text{spec-1} \\
T \Rightarrow \text{error}
\]

Ref.: \( \text{is-intg-val} \text{ 9-3(5)} \)
\( \text{bintg-const} \text{ 8-6(11)} \)

(17) \( \text{mix-init} \text{(init-1, init-2)} = \)
\[
\text{is-\<}(\text{init-1}) \Rightarrow \text{init-2} \\
T \Rightarrow \text{init-1}
\]

7.1.2 ALLOCATION IN AREAS

The aggregate attributes of the based variable to be allocated are evaluated by the instruction \text{eval.alloc-aggr}. This evaluation implies the evaluation of the expressions in the REFER-options which specify extents of the variables.

The generation of the area in which the allocation is made is obtained by evaluating the s-area component, or, if this is not specified, by evaluating the area reference declared with the offset variable specified in the set reference. The set reference is obtained from the s-ptr component, or, if this is not specified, from the pointer reference declared with the based variable. The instruction \text{allocate-area} enters the offset identifying the new generation in the allocation state of the area, and returns the new generation for the use in the instruction \text{initialize}. If the allocation is not possible the AREA condition is raised. After return from the on-unit the area generation is re-evaluated and the allocation is tried again. Immediately after the allocation the pointer value identifying the new generation is assigned to the generation of the set reference, and the integer values of the expressions specified in the REFER-options are assigned to the respective parts of the new generation.
(18) \texttt{eval-alloc-aggr}(aggr) =
\begin{align*}
is-array(aggr) &\implies\nonumber
\texttt{pass}(eva)
\begin{align*}
s-\text{ld}(eva) :& \texttt{eval-alloc-extent}(s-\text{ld}(aggr)), \\
s-\text{ubd}(eva) :& \texttt{eval-alloc-extent}(s-\text{ubd}(aggr)), \\
s-\text{elem}(eva) :& \texttt{alloc-agg}(s-\text{elem}(aggr))
\end{align*}
is-struct(aggr) &\implies\nonumber
\texttt{pass}(eva)
\begin{align*}
\begin{cases}
\forall i \leq \text{length}(aggr) & \texttt{eval-agg}(s-aggr\cdot\text{elem}(i), aggr, eva) \\
\end{cases}
\end{align*}
is-string\text{-da}(aggr) &\implies\nonumber
\texttt{pass}(eva)
\begin{align*}
s-\text{length\cdot da}(eva) :& \texttt{eval-alloc-extent}(s-\text{length\cdot da}(aggr)), \\
en\texttt{pass}(6, aggr, s\text{-init})
\end{align*}
is-area\text{-da}(aggr) &\implies\nonumber
\texttt{pass}(eva)
\begin{align*}
s-\text{size\cdot da}(eva) :& \texttt{eval-alloc-extent}(s-\text{size\cdot da}(aggr)), \\
en\texttt{pass}(6, aggr, s\text{-init})
\end{align*}
T \implies \texttt{eval-aggr}(aggr)
\end{align*}
\text{Ref.:
\begin{align*}
\texttt{eval-aggr} &\ 6-9(23)
\end{align*}
}(19) \texttt{eval-alloc-extent}(extent) =
\begin{align*}
is-ref(extent) &\implies \texttt{eval-intg\cdot expr}(s\cdot expr(extent))
\end{align*}
T \implies \texttt{intg\cdot val}(extent)
\text{Ref.:
\begin{align*}
\texttt{eval-intg\cdot expr} &\ 6-22(60)
\texttt{intg\cdot val} &\ 6-10(27)
\end{align*}
}(20) \texttt{allocate\cdot area}(gen, eva, area) =
\begin{align*}
is\cdot area\cdot gen\cdot da(gen) &\implies \texttt{is-correct\cdot eva}(eva) \implies \texttt{error}
is\cdot free\cdot space(o_1, p_1, AREA) &\implies
\begin{align*}
\texttt{PASS}\cdot mk\cdot gen(eva, o_1, p_1)
\begin{cases}
\texttt{gen-alloc}(o_1, p_1, AREA) \\
\texttt{s\cdot pp}(gen)
\end{cases}
\end{align*}
T \implies
\begin{align*}
\texttt{allocate\cdot area}(gen\cdot 1, eva, area); \\
\texttt{gen-1\cdot eval\cdot ref\cdot gen}(area); \\
\texttt{call\cdot GenP}(AREA)
\end{align*}
\end{align*}
\text{where:
\begin{align*}
o_1 &\ = \text{alloc\cdot space}(\text{alloc\cdot size}(eva), p_1, AREA) \\
p_1 &\ = \text{s\cdot pp}(gen)
\end{align*}
\text{Ref.:
\begin{align*}
\texttt{gen\cdot da} &\ 7-1(53)
\texttt{is\cdot free\cdot space} &\ 3-17(56)
\texttt{el\cdot ass} &\ 3-16(48)
\end{align*}
cont’d
8 7. ALLOCATION, INITIALISATION, AND FREEING OF VARIABLES
Note: $\omega_1$ is the pointer value identifying that part of the area which is reserved on allocation.

$$
(20.1) \setandrefer(ptr, gen, aggr) = 
$$

$$
\text{call-check-cond}(\langle ptr \rangle); \\
\text{initialize-ref}(gen, aggr), \\
\text{convert-assign}(\text{gen}-1, \text{val-op}(\text{PTR}, \text{s-pp}(gen)), \text{area}_1, 0); \\
\text{gen}-1 : \text{eval-ref-gen}(ptr) \\
T \rightarrow \text{error}
$$

where:

$\text{area}_1 = \text{area-expr}(ptr, \#D)$

Ref.:

$$
\text{call-check-cond} 10-22(64) \\
\text{convert-assign} 8-8(16) \\
\text{val-op} 9-9(371) \\
\text{eval-ref-gen} 8-28(82) \\
\text{area-expr} 8-28(81)
$$

$$
(20.2) \text{initialize-ref}(gen, aggr) = 
$$

$$
\text{null}:
\{ \text{convert-assign}(\text{gen}-\_1, \text{op}-\_1, 0, 0) \mid \text{is-ref-s}(aggr) \}
$$

where:

$\text{gen}-\_1 = \text{sub-gen}(gen, \text{eql}-\_1)$

$\text{eql}-\_1 = \text{eval-ql}(\text{is-ref-s}(aggr), aggr)$

$\text{op}-\_1 = \text{val-op}(\text{BINTG-EDA}, \#:s-eva(gen))$

Ref.:

$$
\text{convert-assign} 8-8(16) \\
\text{sub-gen} 8-36(115) \\
\text{eval-ql} 8-31(94) \\
\text{val-op} 9-9(371)
$$

7.1.3 ALLOCATION OF BASED VARIABLES IN MAIN STORAGE

The aggregate attributes of the based variables and the set reference are evaluated as for allocation in areas. The instruction allocate-based enters the pointer value identifying the new generation in the allocation state of the main storage $\S$. The pointer value is also entered into the free-set of the current task to insure freeing of the based storage at task end. The instruction returns the new generation for the use in the instruction initialize.
7.2 ALLOCATION OF AUTOMATIC, STATIC, AND DUMMY VARIABLES

The instruction \texttt{allocate-auto(b,eva)} is used for allocating automatic variables, where \( b \) is the aggregate name of the variable and \( eva \) are its evaluated aggregate attributes. The instruction enters the pointer value identifying the new generation in the allocation stack of the main storage \( S \). The new generation is entered in the aggregate directory \( AG \) under the aggregate name \( b \). The pointer value is entered also in the free-set of the epilogue information \( EI \) to ensure freeing of the automatic storage at block end. The instruction returns the new generation for use in the initializing instruction.

The instruction \texttt{allocate(b,eva,type)} is used for allocating static and dummy variables. It performs the same actions as the instruction \texttt{allocate-auto} except that the new pointer value is not entered in the free set, and that the new generation is entered in \( AG \) on top of the null generation (represented as *).

(24) \texttt{allocate-auto(b,eva) =}

\[\text{is-correct-ev}(eva) \rightarrow \text{error}\]
\[\text{is-free-space}(p_1,S,\text{AUTO}) \rightarrow\]
\[\text{PASS: mk-gen}(eva,p_1)\]
\[S=S; \text{el-alloc}(p_1,S,\text{AUTO})\]
\[S=AG; cb: \langle \text{mk-gen}(eva,p_1) \rangle\]
\[S=EI; s-free-set: s-free-set(EI) \cup \{ b \} \]
\[T \rightarrow \text{sta-overflow}\]

where:
\[p_1 = \text{alloc-space}\left(\text{alloc-size}(eva), S, \text{AUTO}\right)\]

Ref.: \texttt{is-correct-ev} 7-5(9)
\texttt{is-free-space} 3-17(56)
\texttt{mk-gen} 7-5(10)
\texttt{el-alloc} 3-17(55)

Note: \( p_1 \) is the pointer value identifying that part of \( S \) which is reserved on allocation.
IBM LAB VIENNA

31 October 1969

ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

7. ALLOCATION, INITIALISATION, AND FREEING OF VARIABLES 10.1
Note: The storage parts given to components of a variable must be independent of one another.

(35.1) \[ \text{alloc-state} (vr-1) = \text{alloc-state} (vr-2) \]
\[ \text{alloc-space} (z, vr-1, \text{AREA}) = \text{alloc-space} (z, vr-2, \text{AREA}) \]

Ref.: \text{alloc-state} 3-16.1(53)

Note: Storage space reserved on allocation in an area depends only on the size required and on the allocation state of the area.

(37) \[ lbd(eva) = \]
\[ \text{is-array} (eva) \rightarrow s-lbd(eva) \]
\[ \text{is-struct} (eva) \rightarrow 1 \]

(38) \[ ubd(eva) = \]
\[ \text{is-array} (eva) \rightarrow s-ubd(eva) \]
\[ \text{is-struct} (eva) \rightarrow \text{length} (eva) \]

7.3.2 SPECIAL PROPERTIES OF THE STORAGE MAPPING FUNCTION

7.3.2.1 Unaligned string aggregates

Unaligned string aggregates are mapped in a structure-independent way. This property is expressed by the properties of the function \text{str-part}(base, i, j) stated in the following equations. The function \text{str-part} is related to the function \text{map}.

(39) \[ \text{str-part}(base, i, j) = \]

Note: This implementation-defined function gives the pointer value identifying the storage part given to the part of a string aggregate whose base is "base" starting from the ith string element, j string elements long, within the storage given to the entire string aggregate.

(39.1) \[ \text{is-ptr-val} \cdot \text{str-part}(base, i, j) \]

Ref.: \text{is-ptr-val} 3-15(36)

(40) \[ 1 \leq i-1 \leq j \leq 1 \leq j-1 \leq (j - i-1 + 2) \]
\[ \text{is-applic}(\text{str-part}(base, i-1, j-1), \text{alloc-size} \cdot \text{str-eva}(base, j)) \]

Ref.: \text{is-applic} 3-15(37)

(41) \[ \text{size} \cdot \text{str-part}(base, i, j) = \text{alloc-size} \cdot \text{str-eva}(base, j) \]

Ref.: \text{size} 3-15(38)
(42) \[ 1 \leq i-1 \leq i-2 \leq (i-2 + j-2 - 1) \leq (i-1 + j-1 - 1) \Rightarrow \]
\[ \text{str-part}(\text{base},i-2,j-2) = \text{str-part}(\text{base},i-2-1,j-2) \times \text{str-part}(\text{base},i-1,j-1) \]

(43) \[ 1 \leq i-1 \leq (i-1 + j-1 - 1) < i-2 \leq (i-2 + j-2 - 1) \Rightarrow \]
\[ \text{is-indep}(\text{str-part}(\text{base},i-1,j-1), \text{str-part}(\text{base},i-2,j-2)) \]

Ref.: is-indep 3-16(43)

(44) \[ (\text{is-bit-aggr} \lor \text{is-char-aggr})(\text{eva}) \Rightarrow \text{map}(\text{eva},i) = \text{str-part}(\text{base}_0,\text{extent}_1,\text{extent}_2) \]

where:
\[ \begin{align*}
\text{base}_0 &= (\text{is-bit-aggr}(\text{eva}) \to \text{BIT}, \text{is-char-aggr}(\text{eva}) \to \text{CHAR}) \\
\text{extent}_1 &= 1 + \sum_{j=1}^{\text{str-length}(\text{eva})} \text{string-extent} \times \text{aggr-part}(\text{eva},j) \\
\text{extent}_2 &= \text{string-extent} \times \text{aggr-part}(\text{eva},i) 
\end{align*} \]

Ref.: is-bit-aggr 8-34(106) 
is-char-aggr 8-34(107) 
string-extent 8-34(108) 
aggr-part 8-37(119)

(45) \[ \text{str-eva}(\text{base},j) = \mu_0(\text{<da}; \mu_0(\text{<base:base}; \text{<length:length}, \text{<dens:UNAL}))) \]

7.3.2.2 Mapping of picture variables

Picture variables are treated like string variables of corresponding length.

(46) \[ \text{is-pic}\times\text{da}(\text{eva}) \Rightarrow \text{alloc-size}(\text{eva}) = \text{alloc-size}(\text{eva}_1) \]

where:
\[ \begin{align*}
\text{eva}_1 &= \mu(\text{eva}; \text{<da}; \mu_0(\text{<base:base}_1, \text{<length:length}_1))) \\
\text{base}_1 &= (\text{is-bin-pic}\times\text{da}(\text{eva}) \to \text{BIT}, \text{T} \to \text{CHAR}) \\
\text{length}_1 &= \text{str-length}\times\text{da}(\text{eva}) 
\end{align*} \]

Ref.: str-length 9-41(164)

7.3.2.3 Relation between the declared and the allocated size of areas

The following equation expresses the property that an allocation in an area is possible if it is possible in an area with the same allocation state but smaller size.
(47) \[ \text{size}(\text{vr}-1) = \text{alloc-size}(\text{eva}_1) \& \text{size}(\text{vr}-2) = \text{alloc-size}(\text{eva}_2) \& i-2 > i-1 \& \text{alloc-state}(\text{vr}-1) = \text{alloc-state}(\text{vr}-2) \& \text{is-free-space}(p,\text{vr}-1,\text{AREA}) \Rightarrow \text{is-free-space}(p,\text{vr}-2,\text{AREA}) \]

where:
\[
\begin{align*}
\text{eva}_1 &= \mu_0(<s-da:mu_0(<s-size:i-1>),<s-dens:dens>)) \\
\text{eva}_2 &= \mu_0(<s-da:mu_0(<s-size:i-2>),<s-dens:dens>))
\end{align*}
\]

Ref.: size 3-15(33)
alloc-state 3-16.1(53)
is-free-space 3-17(56)
7.3.2.4 Left-to-right equivalence

The following equation expresses the property that the mapping of a structure component depends on the structure attributes only up to and including that component.

\[(48) \quad \text{sub-pp-1}(p, \text{set-eql}(\text{eva}, \text{rl}), \text{eva}) = \text{sub-pp-1}(p, \text{set-eql}(\text{eva}, \text{rl}), \text{left-aggr}(\text{eva}, \text{set-eql}(\text{eva}, \text{rl})))\]

Ref.: sub-pp-1 8-37(117)
set-eql 8-13(31)
left-aggr 8-14(33)

7.4 INITIALIZATION OF VARIABLES

Initialization of variables is accomplished by the instruction

\[\text{initialize}(\text{gen}, \text{aggr}), \text{where gen is the generation of the variable to be initialized, and aggr are the aggregate attributes of the variable.}\]

The distinction is made between the initialization by a call statement, the initialization by a list of initial expressions, and the special initialization of label and entry variables. The latter case in concrete text corresponds to the use of subscripted references as prefixes. In abstract text a label or entry constant appears in this case in the initial attribute together with the subscript list identifying the scalar part to be inserted.

\[(49) \quad \text{initialize}(\text{gen}, \text{aggr}) = \]
\[\text{is-array}(\text{aggr}) \rightarrow \text{initialize}(\text{gen}, \text{s-elem}(\text{aggr}))\]
\[\text{is-struct}(\text{aggr}) \rightarrow \]
\[\text{null};\]
\[\{\text{initialize}(\text{sub-gen}(\text{gen}, \text{rl}_i), \text{s-aggr}\cdot \text{elem}(i, \text{aggr})) \mid 1 \leq i \leq \text{length}(\text{aggr})\}\]
\[\text{is-scalar}(\text{aggr}) \rightarrow \text{initialize-l}(\text{gen}, \text{s-init}(\text{aggr}), \text{s-area}\cdot \text{s-da}(\text{aggr}))\]

where:
\[\text{rl}_i = \text{rep-conc}(\text{dim}\cdot \text{s-eva}(\text{gen}), <\#>, <i>)\]

Ref.: sub-gen 8-36(115)
dim 8-29(86)

\[(50) \quad \text{initialize-l}(\text{gen}, \text{init}, \text{area}) = \]
\[\text{is-call-st}(\text{init}) \& \text{is-2-s-pa-opt}(\text{init}) \rightarrow \text{int-call-st}(\text{init})\]
\[\text{is-init-elem-list}(\text{init}) \rightarrow \text{initial-assign}(\text{gen-list}(\text{gen}), \text{init}, \text{area})\]
\[\text{is-spec-init-elem-list}(\text{init}) \& \]
\[(\text{is-LABEL} \lor \text{is-ENTRY}) \cdot \text{s-da}\cdot \text{base-elem}\cdot \text{s-eva}(\text{gen}) \rightarrow \]
\[\text{special-initialize}(\text{gen}, \text{init})\]
T \rightarrow \text{error}

cont'd

7. ALLOCATION, INITIALISATION, AND FREEING OF VARIABLES 15
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

Ref.: int-call-st 6-13(34)
gen-list 5-11(33)
base-elem 8-29(85)

(51) special-initialize(gen,init) =
    is-<>(init) → null
    T →
    special-initialize(gen, tail(init));
    convert-assgn(gen, op, 0, 0)

where:
    gen = sub-gen(gen, s-sl•head(init))
    op = val-op(da1, s-n•head(init))
    da1 = (is-entry(s-n(init) (AE)) → ENTRY,
           is-label-const (s-n(init) (AE)) → LABEL,
           T → error)

Ref.: convert-assgn 8-8(16)
      sub-gen 8-36(115)
      val-op 9-9(37.1)

(51.1) initial-assgn(genl, init, area) =
    ~is-<>(genl) & is-area•gen-da•head(genl) & (is-<>(init) v is-<>head(init)) →
    initial-assgn(tail(genl), tail(init), area);
    convert-assgn(head(genl), op, 0, 0);
    op: eval-empty
    is-<>(genl) v is-<>(init) → null
    is-init-iter•head(init) →
    initial-assgn(genl, init-1, area);
    init-1: base-rep-pconq (i, s-init•head(init), tail(init));
    i: eval-expr (s-rep•head(init))
    is-*•head(init) → initial-assgn(tail(genl), tail(init), area)
    is-expr•head(init) →
    initial-assgn(tail(genl), tail(init), area);
    convert-assgn(head(genl), op, area, area2);
    op: eval-expr (head(init), da1)
    is-op•head(init) →
    initial-assgn(tail(genl), tail(init), area);
    assign (head(genl), head(op))

where:
    area2 = area-expr(head(init), AE)
    da1 = (is-entry•gen-da•head(genl) → ENTRY,
           T → 0)

Ref.: convert-assgn 8-8(16)
      eval-empty 12-62(15)
      eval-expr 4-22(60)

16 7. ALLOCATION, INITIALISATION, AND FREEING OF VARIABLES
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

8.1.2 SYNTACTIC MODIFICATION OF EXPRESSIONS

The instruction \texttt{mod(expr,eva,i)} returns the modified expression \texttt{expr}, where \texttt{eva} are the controlling evaluated aggregate attributes and \texttt{i} is the step number or, in the case of BY NAME structure expansion, a qualifying identifier. In the latter case, if one of the references in \texttt{expr} does not have a structure component selected by the qualifying identifier, the instruction returns \texttt{R}.

(7.1) \texttt{mod(expr,eva,i)} =

\begin{verbatim}
8 ..Sibling5 Assignment Statement, Expression Evaluation, Reference to Variables 5

\texttt{is-infix-expr-1(expr) \rightarrow}

\texttt{pass-return-mod(expr-1);}
\texttt{s-op-1(expr-1):mod(s-op-1(expr),eva,i),}
\texttt{s-op-2(expr-1):mod(s-op-2(expr),eva,i),}
\texttt{s-opr(expr-1):pass(s-opr(expr))}
\end{verbatim}

\begin{verbatim}
\texttt{is-prefix-expr-1(expr) \rightarrow}

\texttt{pass-return-mod(expr-1);}
\texttt{s-op(expr-1):mod(s-op(expr),eva,i),}
\texttt{s-opr(expr-1):pass(s-opr(expr))}
\end{verbatim}

\begin{verbatim}
\texttt{is-paren-expr-1(expr) \rightarrow}

\texttt{pass-return-mod(expr-1);}
\texttt{s-op(expr-1):mod(s-op(expr),eva,i)}
\end{verbatim}

\texttt{cont'd}
is-ref-1(expr) \rightarrow \text{mod-ref}(expr,eva,i)

is-const(expr) \rightarrow \text{PASS:}expr

is-isub(expr) \rightarrow \text{error}

for: (is-intg-val \land is-id)(i)

Ref.: is-infix-expr-1 8-16(41)
is-prefix-expr-1 8-16(42)
is-Paren-expr-1 8-16(43)
is-ref-1 8-17(44)
is-intg-val 9-3(5)

(9) return-mod(expr) =

is-**s-op-1(expr) \lor is-**s-op-2(expr) \lor is-**s-op(expr) \rightarrow *

T \rightarrow expr

(10) \text{mod-ref}(ref,eva,i) =

is-BUILTIN(attr_0) \rightarrow \text{mod-built-in}(ref,eva,i)

is-var(attr_0) \land is-array(eva_1) \land is-array(eva_1) = \text{dim}(eva) \land s-lbd(eva_1) = s-lbd(eva) \land s-ubd(eva_1) = s-ubd(eva) \rightarrow

\text{PASS:}(\text{ref} <\text{s-s1-insert}(\text{s-s1}(\text{ref}),\text{intg-const}(i))))

is-var(attr_0) \land is-struct(eva_1) \land is-struct(eva) \land \text{length}(eva_1) = \text{length}(eva) \land is-intg-val(i) \rightarrow

\text{PASS:}(\text{ref} <\text{s-id-list}s-id-list(ref) <\text{s-qual\-elem}(i,aggr_1)>>)

is-var(attr_0) \land is-struct(eva_1) \land is-struct(eva) \land is-id(i) \land (\text{sk} (\text{s-qual\-elem}(k,aggr_1) = i)) \rightarrow

\text{PASS:}(\text{ref} <\text{s-id-list}s-id-list(ref) <\text{i}>)

is-var(attr_0) \land is-struct(eva_1) \land is-struct(eva) \land is-id(i) \rightarrow \text{PASS:}*

(is-var(attr_0) \land is-struct(eva_1) \land is-array(eva_1)) \lor
(is-var(attr_0) \land is-scaler(eva_1) \land is-entry(attr_0) \land is-generic(attr_0) \lor
is-label-const(attr_0) \land is-format-const(attr_0) \land is-file-const(attr_0)) \land
is-intg-val(i) \rightarrow

\text{PASS:}ref

T \rightarrow \text{error}

for: (is-intg-val \land is-id)(i)

Ref.: mod-built-in 12-2(1)
dim 8-29(86)
is-intg-val 9-3(5)

(11) \text{intg-const}(i) =

m_{\theta}(<\text{s-da:BI\-NTS-EDA},<\text{s-v:i}>)

6 8. ASSIGNMENT STATEMENT, EXPRESSION EVALUATION, REFERENCE TO VARIABLES
18. **ptr-offset-convert** (eva, op, area-1, area-2) =
   is-PTR*s-da(eva) & is-PTR*op-da(op) v is-offset*s-da(eva) & is-offset*op-da(op) ->
   PASS:op
   is-PR*s-da(eva) & is-offset*op-da(op) & is-var-ref(area-2, AT) ->
   eval-ptr(op, gen);
   gen: eval-ref-gen(area-2)
   is-offset*s-da(eva) & is-PR*op-da(op) & is-var-ref(area-1, AT) ->
   eval-offset(op, gen);
   gen: eval-ref-gen(area-1)
   T --> ERROR

Ref.:  op-da 9-9(37)
       eval-ptr 12-63(157)
       eval-ref-gen 8-28(82)
       eval-offset 12-63(156)

19. **eval-lp-list**(lp) =
    is-<>(lp) -- PASS:<>
    T --
    mk-list(gen, genl);
    genl: eval-lp-list(tail(lp));
    gen: eval-lp(head(lp))

20. **eval-lp**(ref) =
    is-BUILTIN(attr 0) & is-<>s-sl(ref) & is-<>s-ptr(ref) &
    is-<>tail-1<s-ap(ref) --
    eval-ps-lp(id 0, arg-list(ref))
    is-var-ref(ref, AT) --> eval-ref-gen-l(ref)
    T -- ERROR

Ref.:  eval-ps-lp 12-67(170)
       eval-ref-gen-l 8-28(82.1)

(20.1) **is-var-ref**(ref, at) =
    is-ref-l(ref) & is-<>s-ap(ref) & is-var(s-n(ref)(at))

Ref.:  is-ref-l 8-17(44)

(22) **is-var** =
    is-prop-var v is-defined v is-base1

9. ASSIGNMENT STATEMENT, EXPRESSION EVALUATION, REFERENCE TO VARIABLES 9
(23) \texttt{assign}(\texttt{gen}, \texttt{op}) =
\begin{align*}
&\text{is-active-event}(\texttt{gen}, \texttt{pa}) \rightarrow \text{call-cond(\texttt{ERROR})}
\quad \text{is-active}(\texttt{gen}, \texttt{pa}) \rightarrow \text{error}
\quad \text{is-EVENT}\cdot\texttt{gen-da}(\texttt{gen}) \& \text{\texttt{s-compl-op-val}(op) = 1-BIT} \rightarrow \\
&\quad \text{s}=\text{el-ass}(\texttt{s-vr}(\texttt{op}), \texttt{s-pp}(\texttt{gen}), \texttt{s})
\quad \text{\texttt{s}}=\text{activate-tasks}(\texttt{pa})
\quad \text{\texttt{s}}=\text{et}\cdot\texttt{et}<\texttt{gen}>
\quad \Gamma \rightarrow \text{s}=\text{el-ass}(\texttt{s-vr}(\texttt{op}), \texttt{s-pp}(\texttt{gen}), \texttt{s})
\end{align*}

Ref.: is-active-event 5-5(11)
call-cond 10-18(54)
is-active 5-5(12)
gen-la 7-17(53)
\texttt{op-val} 9-9(36)
el-ass 3-16(48)
activate-tasks 5-12(42)

Note: The \texttt{vr}-part of the operand is assigned to that part of \texttt{s} selected by the pointer-part of \texttt{gen}. If an attempt is made to assign to an active event variable, the \texttt{ERROR} condition is raised. If the completion value of a non-active event variable is set to 1-BIT by the assignment, waiting tasks are activated and the event generation is appended to the event trace \texttt{ET}.

8.2 EVALUATION OF EXPRESSIONS

This section defines the pre-evaluation of expressions, the evaluation of scalar expressions in entry-context and in non-entry-context, some special cases of expression evaluation needed in other parts of the definition, and the evaluation of aggregate attributes of expressions.

The evaluation of aggregate expressions in PL/I is reduced to the sequential evaluation of scalar expressions. For this mechanism see the definition of the assignment statement (8.1), the definition of the evaluation of arguments (6.2.3), and the definition of the get and put statement (11.6).

Scalar expressions are evaluated by the instruction \texttt{eval-expr(expr,da)}, where \texttt{expr} is the text of the expression, and the data attribute \texttt{da} specifies the context in which the expression is to be evaluated. The instruction returns an operand. The pre-evaluated expression is evaluated by the instruction \texttt{eval-entry-expr-l} if \texttt{da} is an entry data attribute, otherwise by the instruction \texttt{eval-entry-2}.

(24) \texttt{eval-expr(expr,da) =}
\begin{align*}
&\text{eval-expr-l(expr-1,da);} \\
&\text{expr-1=pre-eval(expr)}
\end{align*}

\texttt{for: (is-entry \& is-Q)(da)
8.2.1 EVALUATION OF POINTER QUALIFIERS AND REFER-OPTIONS

Before the expansion and evaluation of an expression, the generations of proper and based variables referred to in the expression are determined. This pre-evaluation implies the evaluation of pointer qualifiers and REFER-options of based variables. References to proper and based variables are modified in that the generations of the referenced variables are inserted as s-gen component of the text of the references. The pre-evaluation is performed by the instruction \texttt{pre-\text{eval}(expr)}\footnote{which returns the text of the pre-evaluated expression expr.}

8.2.1.1 Pre-evaluation of expressions

Pre-evaluated are those references in an expression which are relevant for the expansion of aggregate expressions, i.e. references in subscript expressions, in arguments of user defined functions, and in non-expanding arguments of built-in functions are ignored for the pre-evaluation.

\begin{align*}
(25.1) \text{pre-\text{eval}(expr)} &= \\
&= \begin{cases} \\
\text{is-infix-expr(expr) \rightarrow} & \\
\text{pass(expr-1):} & \text{pre-\text{eval}(s-op-1(expr))}, \\
\text{pre-\text{eval}(s-op-2(expr))}, \\
\text{pass(s-opr(expr-1)):} & \text{pass(s-opr(expr))} \\
\text{is-prefix-expr(expr) \rightarrow} & \\
\text{pass(expr-1):} & \text{pre-\text{eval}(s-opr(expr))}, \\
\text{pass(s-opr(expr-1)):} & \text{pass(s-opr(expr))} \\
\text{is-paren-expr(expr) \rightarrow} & \\
\text{pass(expr-1):} & \text{pre-\text{eval}(s-op(expr))} \\
\text{is-ref(expr) \rightarrow} & \text{pre-\text{eval-ref(expr)}} \\
\text{(is-isub \& is-const)(expr) \rightarrow} & \text{PASS: expr} \\
\end{cases}
\end{align*}
(27) \texttt{pre-eval-ref(ref) =} \\
\texttt{is-BUILTIN(attr)} \rightarrow \\
\texttt{pass(ref-1);} \\
\texttt{s-ap(ref-1):pre-eval-ap(s-ap(ref),id_0);} \\
\texttt{ref-1:pass(ref)} \\
\texttt{is-based(attr)} \rightarrow \\
\texttt{pass(ref-1);} \\
\texttt{s-gen(ref-1):pass-op-gen(eva,p);} \\
\texttt{eva:eval-based-aggr(agr_0,p,eq1);} \\
\texttt{p:pass-op-val(op);} \\
\texttt{op:eval-ptr-ref(ref);} \\
\texttt{ref-1:pass(ref)} \\
\texttt{is-prop-var(attr)} \& \texttt{is-gen(gen)} \rightarrow \texttt{PASS:(ref:<s-gen:gen>)} \\
\texttt{is-prop-var(attr)} \rightarrow \texttt{error} \\
\texttt{T \rightarrow PASS:ref} \\

where: \\
\texttt{gen = head(n_0(DH)(AD))} \\

Ref.: \\
\texttt{mk-gen 7-5(10)} \\
\texttt{op-val 9-9(36)} \\
\texttt{is-gen 3-14(30)} \\

Note: For based references a new generation is formed from the evaluated aggregate attributes of the based variable, and the pointer value resulting from the evaluation of the pointer qualifier. The evaluation of the aggregate attributes of based variables, which implies the evaluation of \texttt{REFER}-options, is performed by the instruction \texttt{eval-based-aggr} (cf. 8.2.1.2). For proper variables the most recent generation is taken from the aggregate directory. Expanding arguments to built-in functions are pre-evaluated by the instruction \texttt{pre-eval-ap}.

(28) \texttt{pre-eval-ap(ap,id) =} \\
\texttt{is-<>(ap) \rightarrow PASS::<>} \\
\texttt{length(ap) > 1 \rightarrow error} \\
\texttt{is-<>head(ap) \rightarrow PASS::<><>} \\
\texttt{T \rightarrow} \\
\texttt{pass(ap-1);} \\
\{ \texttt{elem(i) \& elem(1)} (ap-1):pre-eval-ap(elem(i,head(ap)),id,i) \mid \\
1 \leq i \leq \text{length*head(ap)} \}
8.1.1.1 Evaluation of Pointer Qualifiers

The pointer qualifier of a based reference ref is evaluated by the instruction `eval-plq-ref(ref)`, which returns the resulting operand. If no explicit pointer qualifier is present, the pointer reference contained in the declaration of the based variable is evaluated. If the pointer qualifier is the reference to an offset variable, the resulting offset operand is converted to a pointer operand by the instruction `ptr-conv(ptr,area)`. area is the reference to an area variable if contained in the declaration of the offset variable, or otherwise.

Note: The attributes aggr are used to interpret the name-qualification list contained in the REFER-option.

(36) \[ \text{eval-refer-options}(\text{aggr-1}, \text{gen}, \text{aggr}) = \]

\[ \text{is-array}(\text{aggr-1}) \rightarrow \]

\[ \text{pass}(\text{mi}); \]

\[ \text{s-lbd}(\text{mi}): \text{eval-extent}(\text{s-lbd}(\text{aggr-1}), \text{gen}, \text{aggr}), \]

\[ \text{s-ubd}(\text{mi}): \text{eval-extent}(\text{s-ubd}(\text{aggr-1}), \text{gen}, \text{aggr}), \]

\[ \text{s-elem}(\text{mi}): \text{eval-refer-options}(\text{s-elem}(\text{aggr-1}), \text{gen}, \text{aggr}) \]

\[ \text{is-struct}(\text{aggr-1}) \rightarrow \]

\[ \text{pass}(\text{mi}); \]

\[ \{ \text{s-aggr\{elem(i)\}(\text{mi})}: \text{eval-refer-options}(\text{s-aggr\{elem(i,aggr-1)\}, \text{gen}, \text{aggr}) \mid 1 \leq i \leq \text{length} (\text{aggr-1}) \} \]

\[ \text{is-string}\{\text{s-da}(\text{aggr-1}) \rightarrow \]

\[ \text{pass}(\text{mi}); \]

\[ \text{s-length}\{\text{s-da}(\text{mi})}: \text{eval-extent}(\text{s-length}\{\text{s-da}(\text{aggr-1}), \text{gen}, \text{aggr} \}; \]

\[ \text{mi}: \text{pass} (6 \{\text{aggr-1:s-init}\}) \]

\[ \text{is-area}\{\text{s-da}(\text{aggr-1}) \rightarrow \]

\[ \text{pass}(\text{mi}); \]

\[ \text{s-size}\{\text{s-da}(\text{mi})}: \text{eval-extent}(\text{s-size}\{\text{s-da}(\text{aggr-1}), \text{gen}, \text{aggr} \}; \]

\[ \text{mi}: \text{pass} (6 \{\text{aggr-1:s-init}\}) \]

\[ \text{T} \rightarrow \text{PASS}_{\mu}(\langle \text{s-da:eval-fa-l\{s-da}(\text{aggr-1})\rangle, \langle \text{s-dens:s-dens}(\text{aggr-1})\rangle) \]

Ref.: eval-da-1 6-10(26)

(37) \[ \text{eval-extent}(\text{extent}, \text{gen}, \text{aggr}) = \]

\[ \text{is-refer}(\text{extent}) \rightarrow \]

\[ \text{pass-op-val}(\text{op-1}); \]

\[ \text{op-1: convert-1}(\text{aggr-scal}(\text{BING-EDA}, \text{op}); \]

\[ \text{op: gen-op}(\text{gen-1,s}); \]

\[ \text{gen-1: pass-sub-1eg}(\text{gen, eval-ql(\text{s-refer(extent)}, \text{aggr}))} \]

\[ \text{T} \rightarrow \text{pass-intg-val}(\text{extent}) \]

Ref.: op-val 9-9(36)

\[ \text{convert-1} 9-29(119) \]

\[ \text{sub-}
\[ \text{eval-ql} 8-31(94) \]

\[ \text{intg-val} 6-10(27) \]

8. ASSIGNMENT STATEMENT, EXPRESSION EVALUATION, REFERENCE TO VARIABLES 15
(37.1) \texttt{eval-ptr-ref(ref) =}
\begin{verbatim}
    ptr-conv(op,area-expr(ptr_1,Δ2));
    op:eval-ref(ptr_1)
\end{verbatim}

where:
\begin{align*}
    \text{ptr}_1 &= (\text{is-ω*ptr(ref) → s.ptr(ref)}, \\
    & \quad \text{is-ω*ptr(attr}_0) → s.ptr(attr}_0), \\
    & \quad T → \text{error}
\end{align*}

(39) \texttt{ptr-conv(op,area) =}
\begin{verbatim}
    is-PTR•op-3a(op) → PASS:op \\
    is-offset•op-3a(op) & is-var-ref(area,Δ2) → \\
    eval-ptr(op,gen); \quad gen:eval-ref-gen(area) \\
    T → \text{error}
\end{verbatim}

Ref.:
\begin{align*}
    \text{op-3a} & \quad 9-9(37) \quad 168.8 \\
    \text{is-var-ref} & \quad 8-9(20.1) \quad \text{ASSIGNMENT STATEMENT, EXPRESSION EVALUATION, REFERENCE TO VARIABLES}
\end{align*}

8.2.1.4 Abstract Syntax of Pre-Evaluated Expressions

The pre-evaluation of expressions changes the abstract syntax of the expressions, since references to variables may contain in the component s-gen the current generation of the variable. The abstract syntax of pre-evaluated expressions is given by the predicate \texttt{is-expr-1}.

(40) \texttt{is-expr-1 =}
\begin{verbatim}
    is-infix-expr-1 v is-prefix-expr-1 v is-paren-expr-1 v is-ref-1 v is-const v is-isub v is-ismub-ival
\end{verbatim}

(41) \texttt{is-infix-expr-1 =}
\begin{verbatim}
    (<s-opr:is-infix-opr>, <s-op-1:is-expr-1>, <s-op-2:is-expr-1>)
\end{verbatim}

(42) \texttt{is-prefix-expr-1 =}
\begin{verbatim}
    (<s-opr:is-prefix-opr>, <s-op:is-expr-1>)
\end{verbatim}

(43) \texttt{is-paren-expr-1 =}
\begin{verbatim}
    (<s-op:is-expr-1>)
8.2.2 EVALUATION OF EXPRESSIONS IN ENTRY-CONTEXT

Scalar expressions in a context which expects an entry operand are evaluated by the instruction `eval-entry-expr(ref, descr)`, where `ref` is the text of the expression and `descr`, if present, is a list of descriptors for determining the generic selection for the case that `ref` is a generic reference. The instruction returns an operand. Expressions in entry-context are right-part expressions of assignment statements where the left-part references are references to entry variables, or argument expressions with corresponding entry parameters.

(47) \[ \text{eval-entry-expr}(ref, descr) = \]

\[ \text{eval-entry-expr}(expr-1, descr) ; \]

\[ \text{expr-1: pre-eval}(ref) \]

for: is-expr(ref)

Note: The pre-evaluation of expressions by the instruction \texttt{pre-eval} is described in 8.2.1.
(47.1) \( \text{eval-entry-expr-l}(\text{ref}, \text{descr}) = \)

\[
\begin{align*}
\text{is-\textbf{is-paren-expr-l}(\text{ref})} & \rightarrow \text{eval-entry-\textbf{is-paren-expr-l}(s-op(\text{ref})}, \text{descr}) \\text{\textbf{-}} \text{is-ref-l(\text{ref})} & \rightarrow \textbf{error} \\text{is-var(\text{attr}_0)} & \& \text{is-entry-s-da(\text{aggr}_1)} \rightarrow \\
\text{\textbf{eval-entry}(n,s-da(\text{aggr}_1), s-ap(\text{ref}), \text{ref}); } \text{op: eval-var-ref(\text{ref})} \\text{is-entry(\text{attr}_0)} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\text{is-entry(\text{attr}_0)} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\text{is-entry(\text{attr}_0)} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\text{is-entry(\text{attr}_0)} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\
\text{\textbf{eval-entry}(n_0, \text{attr}_0, s-ap(\text{ref}), \text{ref})} \\text{is-generic(\text{attr}_0)} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\text{is-generic(\text{attr}_0)} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\text{is-generic(\text{attr}_0)} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\text{is-generic(\text{attr}_0)} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\
\text{\textbf{eval-entry}(\mu(\text{ref}_3, \text{s-ap}: \text{s-ap}(\text{ref}_3, \text{s-ap}(\text{ref}_3, \text{ref})))}, \text{\textbf{error}}) \\text{is-\textbf{is-entry-s-sl}(\text{ref})} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\text{is-\textbf{is-entry-s-sl}(\text{ref})} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\text{is-\textbf{is-entry-s-sl}(\text{ref})} & \& \text{is-\textbf{is-ref-l}(\text{ref})} & \rightarrow \text{\textbf{error}} \\
\text{\textbf{eval-entry}(\text{ref}_3, \text{attr}_0, s-ap(\text{ref}_3), \text{ref}); } \text{op: eval-var-ref(\text{ref})} \\text{is-generic-sel(\text{attr}_0, \text{descr}_1)} & \rightarrow \text{\textbf{error}} \\text{\textbf{error}} \end{align*}
\]

\(\text{where:}\\\text{ref}_3 = \text{generic-sel(\text{attr}_0, \text{descr}_1)}\\\text{descr}_1 = (\text{is-\textbf{is-ref-l}(\text{ref})} \rightarrow \text{\textbf{error}}, \text{\textbf{error}})\)

\(\text{for: is-\textbf{is-ref-l}(\text{ref})}\\\text{Is-\textbf{is-ref-l}(\text{ref})} = \text{6-35(96)}\)

Ref.: \(\text{is-var 8-9(22)}\) \(\text{Op-val 9-9(36)}\)
\(\text{is-float-generic-builtin 12-11(26)}\)
\(\text{built-in-sel 12-11(27)}\)
\(\text{un-name 3-10(20)}\)
\(\text{generic-sel 6-31(77)}\)
\(\text{attr-expr-list 6-35(96)}\)

Note: \(\text{Generic selection is described in 6.2.5.}\)

(47.2) \( \text{eval-entry}\!(n, l, a, p, r) = \)

\[
\begin{align*}
\text{is-\textbf{is-\textbf{is-ref-l}(\text{ap})} & \& \text{is-entry\textbf{-s-da\text{-s-ret-type}(da)} \rightarrow \text{\textbf{error}}} \\text{is-entry(\text{da}) & \& \text{is-\textbf{is-ref-l}(\text{ap}) \rightarrow} & \text{eval-entry}(n-1, s-da, s-ret-type(da), tail(ap), \text{ref}); } \text{n-1: \text{\textbf{pass-op-val}(op)}; } \text{op: eval-function(n, head(ap), l, r) \rightarrow \textbf{error}} \\
\text{op-val 9-9(36); } \text{eval-function 6-13(35)}
\end{align*}
\]

18 8. \(\text{Assignment Statement, Expression Evaluation, Reference to Variables}\)
(54.1) \( \text{eval-ref-1}(\text{ref}) = \)
\[
is\-\text{var}(\text{attr}_0) \& \text{is-}\text{-}\text{entry\-s\-da}(\text{aggr}_1) \implies \\
\quad \text{eval\-entry-ref}(\text{n}, \text{s\-da}(\text{aggr}_1), \text{s\-ap}(\text{ref}), \text{ref}) ; \\
\quad \text{n}: \text{PASS: op\-val}(\text{op}) ; \\
\quad \text{op}: \text{eval\-var\-ref}(\text{ref}) \\
\quad \text{is\-var\-ref}(\text{ref}, \text{AT}) \implies \text{eval\-var\-ref}(\text{ref}) \\
\text{is\-entry}(\text{attr}_0) \& \text{is\-}\text{\-s\-sl}(\text{ref}) \& \text{is\-}\text{-}\text{q\-s\-ptr}(\text{ref}) \implies \\
\quad \text{eval\-entry-ref}(\text{n}_0, \text{attr}_0, \text{s\-ap}(\text{ref}), \text{ref}) \\
\quad \text{is\-generic}(\text{attr}_0) \& \text{is\-}\text{-}\text{\-s\-sl}(\text{ref}) \& \text{is\-}\text{-}\text{q\-s\-ptr}(\text{ref}) \implies \\
\quad \text{eval\-ref}(\mu(\text{ref}_0; \text{s\-ap}\text{s\-ap}(\text{ref}_0) \text{\text{-}\text{ap}(\text{ref)}))} \\
\quad \text{is\-BUILTIN}(\text{attr}_0) \& \text{is\-}\text{-}\text{\-s\-sl}(\text{ref}) \& \text{is\-}\text{-}\text{q\-s\-ptr}(\text{ref}) \& \\
\quad \text{is\-}\text{-}\text{\-\-tail\-i\-s\-ap}(\text{ref}) \implies \\
\quad \text{eval\-builtin}(\text{id}_0, \text{arg\-list\-s\-ap}(\text{ref})) \\
\quad (\text{is\-label\-const} \lor \text{is\-}\text{-format\-const} \lor \text{is\-}\text{-file\-const})(\text{attr}_0) \& \text{is\-}\text{-}\text{\-s\-sl}(\text{ref}) \& \\
\quad \text{is\-}\text{-}\text{-\-s\-ap}(\text{ref}) \& \text{is\-}\text{-}\text{-q\-s\-ptr}(\text{ref}) \implies \\
\quad \text{PASS: val\-op}(\text{da}_0, \text{n}_0). \\
\]
\[ \text{T} \implies \text{ERROR} \]

Where:
\[ \text{ref}_0 = \text{generic\-sel}(\text{attr}_0, \text{descrl}_1) \]
\[ \text{descrl}_1 = \text{attr\-expr\-list}(\text{arg\-list\-s\-ap}(\text{ref}), \text{AT}) \]
\[ \text{da}_0 = (\text{is\-label\-const} \lor \text{is\-}\text{-format\-const}(\text{attr}_0) \implies \text{LABEL}, \text{is\-}\text{-file\-const}(\text{attr}_0) \implies \text{FILE}) \]

Ref.:
- is\-var 6-9(22)
- op\-val 9-9(36)
- is\-var\-ref 8-9(20.1)
- eval\-builtin 12-3(3)
- arg\-list 8-5(6)
- val\-op 9-9(37.1)
- generic\-sel 6-31(77)
- attr\-expr\-list 6-35(96)

Note: Generic selection is described in 6.2.5.

(56) \( \text{eval\-entry-ref}(\text{n}, \text{da}, \text{ap}, \text{ref}) = \)
\[
\quad \text{eval\-entry-ref-1}(\text{op}, \text{s\-da}\text{s\-ret\-type}(\text{da}), \text{tail\-1}(\text{ap}), \text{AT}); \\
\quad \text{op}: \text{eval\-function}(\text{n}, \text{da}, \text{arg\-list}(\text{ap}), \text{ref}) \\
\]

Ref.:
- eval\-function 6-13(35)
- arg\-list 8-5(6)
(57) \[
\text{eval-entry-ref-1(op, da, ap) =}
\]
\[
is-ENTRY\cdot op-da(op) \rightarrow \text{eval-entry-ref(op-val(op), da, ap, 0)}
\]
\[
is-\times\langle ap\rangle \rightarrow \text{PASS:op}
\]
\[
T \rightarrow \text{ERROR}
\]

Ref.: \(\text{op-da 9-9(37)}\)
\(\text{op-val 9-9(36)}\)

Note: An entry value is always called as a function. If the argument part is exhausted, the function is given the empty argument list.

(58) \[
\text{eval-var-ref(ref) =}
\]
\[
\text{gen-op(gen) ;}
\]
\[
\text{gen:eval-ref-gen-1(ref)}
\]

Ref.: \(\text{eval-ref-gen-1 5-28(82.1)}\)

(59) \[
\text{gen-op(gen) =}
\]
\[
is-scalar\cdot s-eva(gen) \rightarrow \text{PASS:}\mu_0(\langle s-eva:s-eva(gen), s-evr:s-pp(gen)\rangle)
\]
\[
T \rightarrow \text{ERROR}
\]

8.2.4 SPECIAL CASES OF EXPRESSION EVALUATION

(60) \[
\text{eval-intg- expr(expr) =}
\]
\[
(is\times op \vee is-0)\cdot expr \rightarrow \text{PASS: expr}
\]
\[
is-expr(expr) \rightarrow
\]
\[
\text{pass-op-val(op)};
\]
\[
op:convert-1(agr-scalar(BINTC-EDA), op-1);
\]
\[
op-1:eval-expr(expr, 0)
\]
\[
T \rightarrow \text{ERROR}
\]

for: \((is-expr \vee is-refer \vee is\times op \vee is-0)\cdot expr\)

Ref.: \(\text{op-val 9-9(36)}\)
\(\text{convert-1 9-29(119)}\)

Note: The instruction is used when the value of an expression is to be converted to an integer value.
by *.

(68) \text{delete-length}(\text{da}) =
\begin{align*}
\text{is-string} (\text{da}) & \rightarrow \mu (\text{da} ; < \text{s-length} : \star>) \\
\text{T} & \rightarrow \text{da}
\end{align*}

(69) \text{aggr-prefix-expr}(\text{aggr}, \text{opr}) =
\begin{align*}
\text{is-array} (\text{aggr}) & \rightarrow \mu (\text{aggr} ; < \text{s-elem} : \text{aggr-prefix-expr}(\text{s-elem} (\text{aggr}), \text{opr})>) \\
\text{is-struct} (\text{aggr}) & \rightarrow \\
\text{List} & \rightarrow \\
\mu_0 (\text{s-aggr} : \text{aggr-prefix-expr}(\text{s-aggr} \cdot \text{elem}(i, \text{aggr}), \text{opr}))) \\
\text{is-scalar} (\text{aggr}) & \rightarrow \text{aggr-scalar} \cdot \text{da-prefix}(\text{s-da} (\text{aggr}), \text{opr})
\end{align*}

(70) \text{da-prefix}(\text{da}, \text{opr}) =
\text{eda-prefix}(\text{delete-length} (\text{da}), \text{opr})

Ref.: \text{eda-prefix} 9-28(117)

(70.1) \text{aggr-ref}(\text{ref}, \text{at}) =
\begin{align*}
\text{(is-var)} (\text{attr}_0) & \rightarrow \text{aggr-var} (\text{aggr}, \text{s-ap} (\text{ref})) \\
\text{is-entry-const} (\text{attr}_0) & \rightarrow \text{aggr-scalar-ref} (\text{aggr-scalar} (\text{attr}_0), \text{s-ap} (\text{ref})) \\
\text{is-file-const} (\text{attr}_0) & \rightarrow \text{aggr-scalar} (\text{FILE}) \\
\text{(is-label-const} \lor \text{is-format-const}) (\text{attr}_0) & \rightarrow \text{aggr-scalar} (\text{LABEL}) \\
\text{is-generic} (\text{attr}_0) & \rightarrow \text{aggr-generic} (\text{ref-g}_0, \text{s-ap} (\text{ref}), \text{at}) \\
\text{is-BUILTIN} (\text{attr}_0) & \rightarrow \text{aggr-builtin} (\text{id}_0, \text{arg-list} \cdot \text{s-ap} (\text{ref}), \text{at})
\end{align*}

where:
\begin{align*}
\text{attr}_0 & = \text{s-n} (\text{ref}) (\text{at}) \\
\text{aggr}_1 & = \text{sub-aggr} (\text{aggr}, \text{attr}_0 , \text{s-sl} (\text{ref}), \text{eql}_1) \\
\text{eql}_1 & = \text{eval-ql} (\text{tail} \cdot \text{e-id-list} (\text{ref}), \text{aggr}_0) \\
\text{ref-g}_0 & = \text{generic-sel} (\text{attr}_0, \text{arg-list}_1) \\
\text{arg-list}_1 & = \text{attr-expr-list} (\arg-list \cdot \text{s-ap} (\text{ref}), \text{at})
\end{align*}

Ref.: \text{is-var} 8-3(22) \\
\text{aggr-builtin} 12-7(18) \\
\text{arg-list} 8-5(6) \\
\text{eval-ql} 8-31(94) \\
\text{generic-sel} 6-11(77) \\
\text{attr-expr-list} 6-35(96)

Note: If ref has no argument part, the aggr-ref just returns aggr_1.
(70.2) \text{aggr-var}(\text{aggr}, \text{ap}) =
\begin{align*}
\text{is-array}(\text{aggr}) &\rightarrow \mu(\text{aggr}; (s\text{-elem}: \text{aggr-var}(s\text{-elem}(\text{aggr}), \text{ap}))) \\
\text{length}(\text{aggr}) &\rightarrow \sum_{i=1}^{\text{length}(\text{aggr})} (s\text{-aggr}:: \text{aggr-var}(s\text{-aggr}\text{-elem}(i, \text{aggr}), \text{ap})) \\
\text{is-struct}(\text{aggr}) &\rightarrow \text{aggr}\text{-scalar-ref}(\text{aggr}, \text{ap})
\end{align*}

(70.3) \text{aggr-scalar-ref}(\text{aggr}, \text{ap}) =
\begin{align*}
\text{is-<>}(\text{ap}) &\rightarrow \text{aggr} \\
\text{is-entry\text{-s-da}(aggr)} &\rightarrow \text{aggr-scalar-ref}(s\text{-ret-type\text{-s-da}(aggr)}\text{-tail}(\text{ap})) \\
T &\rightarrow \text{error}
\end{align*}

(70.4) \text{aggr-generic}(\text{ref}, \text{ap}, \text{at}) =
\text{aggr-scalar-ref}(\text{aggr-ref}(\text{ref}, \text{at}), \text{ap})

(75) \text{sub-aggr}(\text{aggr}, \text{sl}, \text{eql}) =
\begin{align*}
\text{is-<>-list}(\text{sl}) &\& \text{is-<>}(\text{eql}) \rightarrow \text{aggr} \\
\text{is-array}(\text{aggr}) &\& (\text{is-<>}(\text{sl}) \lor \text{is-<>}(\text{eql})) \rightarrow \\
\mu(\text{aggr}; (s\text{-elem}: \text{sub-aggr}(s\text{-elem}(\text{aggr}), \text{tail}(\text{sl}), \text{eql}))) \\
\text{is-array}(\text{aggr}) &\rightarrow \text{sub-aggr}(s\text{-elem}(\text{aggr}), \text{tail}(\text{sl}), \text{eql}) \\
\text{is-struct}(\text{aggr}) &\& \text{is-<>}(\text{eql}) \rightarrow \\
\text{sub-aggr}(s\text{-aggr}\text{-elem}(\text{head}(\text{eql}), \text{aggr}), \text{sl}, \text{tail}(\text{eql})) \\
T &\rightarrow \text{error}
\end{align*}

8.2.5.2 Evaluated attributes of array expressions

For the syntactic expansion of array expressions it is necessary to known
array-bounds before the actual evaluation. The instruction \text{array-eva-expr}(\text{expr})
returns the aggregate attributes of the result of the expression with evaluated
array bounds, where \text{expr} is the pre-evaluated expression. For the pre-evaluation
of expressions see 8.2.1.
(76) \texttt{array-eva-infix (expr) =}
\begin{align*}
\text{-is-array-aggr-expr}(expr, \text{AT}) & \rightarrow \text{PASS:}^* \\
\text{is-infix-expr-1}(expr) & \rightarrow \\
\text{PASS:array-eva-infix}(eva-1, eva-2) ; \\
eva-1: & \text{array-eva-infix}(s-op-1(expr)), \\
eva-2: & \text{array-eva-infix}(s-op-2(expr)) \\
\end{align*}

(is-prefix-expr-1 \lor is-paren-expr-1) (expr) \rightarrow \text{array-eva-infix}(s-op(expr))

\text{is-prop-var}(\text{attr}_0) \lor is-based(\text{attr}_0) \rightarrow

\text{PASS:sub-aggr}(s-eva\cdot s-gen(expr), s-sl(expr), eql_1)

\text{is-defined}(\text{attr}_0) \rightarrow \text{PASS:sub-aggr}(\text{den}_0, s-sl(expr), eql_1)

\text{is-BUILTIN}(\text{attr}_0) \rightarrow

\text{array-eva-built-in}(\text{head}\cdot s-id-list(expr), \text{arg-list}\cdot s-ap(expr), 1)

T \rightarrow \text{ERROR}

where:

\begin{align*}
\text{attr}_0 &= s-n(expr) (\text{AT}) \\
\text{den}_0 &= s-n(expr) (\text{QU}) \\
eql_1 &= \text{eval-ql}(\text{tail}\cdot s-id-list(expr), s-aggr(\text{attr}_0))
\end{align*}

Ref.: \text{arg-list 8-5(6)}
\text{eval-ql 8-31(94)}

(77) \text{array-eva-infix}(eva-1, eva-2) =
\begin{align*}
\text{-*(eva-2)} & \rightarrow eva-1 \\
\text{-*(eva-1)} & \rightarrow eva-2 \\
\text{dim}(eva-1) = \text{dim}(eva-2) & \rightarrow ^* \\
\text{dim}(eva-1) = \text{dim}(eva-2) \lor \text{s-lbd}(eva-1) = \text{s-lbd}(eva-2) \rightarrow \\
\text{s-ubd}(eva-1) = \text{s-ubd}(eva-2) & \\
\mu(eva-1; \langle s-elem: array-eva-infix(s-elem(eva-1), s-elem(eva-2)) \rangle)
\end{align*}

T \rightarrow \text{ERROR}

Ref.: \text{dim 8-29(86)}

(78) \text{array-eva-built-in}(id, expr-list, i) =
\begin{align*}
\text{-<>(expr-list)} & \rightarrow \text{PASS:}^* \\
T & \rightarrow \\
\text{PASS:array-eva-infix}(eva-1, eva-2) ; \\
eva-1: & \text{array-eva-built-in-arg}(id, head(expr-list), i), \\
eva-2: & \text{array-eva-built-in}(id, tail(expr-list), i + 1)
\end{align*}

where:

\begin{align*}
i_1 = \{ \text{id} \in (\text{mk-id}(\text{MAX}), \text{mk-id}(\text{MIN})) \rightarrow 1, \\
T \rightarrow i \}
\end{align*}
8.2.5.3 Attributes of scalar references in non-entry context

An entry value resulting from the evaluation of a reference in non-entry context is always invoked as a function. The function `final-ret-type` returns the attributes of the final result.

```
(80) final-ret-type(aggr) =
    is-entries-aggr(aggr) -> final-ret-types-ret-types-da(aggr)
    T -> aggr
```

Note: The function returns the area reference declared with the offset if the expression is an offset expression, T otherwise. The area reference is used for implicitly converting an offset operand to a pointer operand.

8.3 Evaluation of References to Variables

The generation referenced by a reference `ref` to a variable is evaluated by the instruction `eval-ref-gen(ref)`. The referenced variable may be a proper variable of any storage class, a defined variable, or a based variable.

```
(82) eval-ref-gen(ref) =
    eval-ref-gen-1(ref-1);
    ref-1: pre-eval(ref)
```

Ref.: pre-eval 8-11(25.1)

Note: For pre-evaluation of references see 8.2.1.

```
(82.1) eval-ref-gen-1(ref) =
    is-ref-1(ref) &
    (lengths-sl(ref) = 0 v lengths-sl(ref) = ref-dim(aggr, egl)) ->
    eval-sub-gen(ref, rl);
    rl: d-rl(s-sl(ref), egl, eva)
    T -> ERROR
```

Ref.: is-ref-1 8-17(44)
Note: With the help of the function ref-dim the correct length of the subscript-list is tested.
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

(94) \[ \text{eval-ql}(ql, aggr) = \]
\[ \text{is-<>}(ql) \rightarrow <> \]
\[ \text{is-array}(aggr) \rightarrow \text{eval-ql}(ql, s-elem(aggr)) \]
\[ \text{is-struct}(aggr) \rightarrow \langle eq_i \rangle \text{eval-ql}(\text{tail}(ql), s-aggr\cdot elem(eq_i, aggr)) \]
\[ T \rightarrow \text{error} \]

where:
\[ s_{i1} = (v_i) (\text{head}(ql) = s-qual\cdot elem(i, aggr)) \]

8.3.2 REFERENCE TO DEFINED VARIABLES

Rule 6.1: _isub-defined variables_

The number of dimensions of the defined variable determines the number of _isub_ variables to be created. The values of the _isub_ variables are determined by the evaluated subscripts of the reference to the defined variable. The values are entered as _s-v_ component of the references to the _isub_ variables in the base reference by the function _insert-isubs_. The base reference, modified in this way, is evaluated and the resulting generation is treated like in the simple-defined case. That part of the reference list not used to give values to _isubs_ is used to determine the appropriate sub-generation.

(94.1) \[ \text{eval-isub-gen}(ref, rl) = \]
\[ \text{length}(rl) \geq \dim(aggr) \land \text{is-non-varying}(aggr) \land \]
\[ \text{is-var-ref}(s-base(attr), AT) \land \text{is-prop-var}(attr-b) \land \]
\[ \text{is-corrsp}(base-elem(aggr), aggr-b) \land \text{is-0*pos}(attr) \rightarrow \]
\[ \text{base-sub-gen}(gen-1, rest-rl_1); \]
\[ \text{gen-1:}\text{base-simple-ref-gen}(gen-2, base-elem(\text{eval})); \]
\[ \text{gen-2:}\text{eval-ref-gen}(\text{insert-isubs}(s-base(attr), isub-rl_1)) \]
\[ T \rightarrow \text{error} \]

where:
\[ attr-b = s-n\cdot s-base(attr), AT \]
\[ aggr-b = aggr-ref(s-base(attr), AT) \]
\[ \text{length}(rl) \]
\[ rest-rl_1 = \text{LIST elem}(i, rl) \]
\[ i = k+1 \]
\[ i = 1 \]
\[ isub-rl_1 = \text{LIST elem}(i, rl) \]
\[ i = 1 \]

Ref.: \[ \text{is-var-ref} 8-9(20.1) \]
\[ \text{is-corrsp} 6-20(51) \]
\[ aggr-ref 8-25(70.1) \]

(97) \[ \text{insert-isubs}(ref, list) = \]
\[ \mu(\text{ref}:<s-\text{v*}: \text{isub-val}(s-i\cdot s(ref), list) >: \text{is-\text{isubs}(ref)}) \]

8. ASSIGNMENT STATEMENT, EXPRESSION EVALUATION, REFERENCE TO VARIABLES 31
The discrimination between the two cases is made by the predicates is-corresp and is-overlay. In the simple-defined case the pointer-part and the mapping information of the base generation are retained, and only the eva-part is replaced by the evaluated aggregate attributes of the defined variable. The eva-part and the mi-part of the resulting generation then may differ in array-bounds and string-lengths. In the overlay-defined case a new generation is formed from the base generation by determining a single pointer value identifying the referenced part of the base, which forms the pointer part of the new generation, and installing the evaluated aggregate attributes of the defined variable as the eva-part and the mi-part of the new generation.

```
(100) eval-def-gen(ref) =
    is-corresp(aggr,aggr-b) & is-0*is-pos(attr) -->
    pass-simple-def-gen(gen,eva)
        gen:eval-ref-gen(s-base(attr))
    is-overlay(attr,AT) -->
    pass-overlay-gen(gen,eva,i)
        gen:eval-ref-gen(s-base(attr)),
        i:eval-int expr(s-pos(attr))
    T -- error

where:
    aggr-b = aggr-ref(s-base(attr),AT)

Ref.:  is-corresp 6-20(51)
       eval-int expr 6-22(60)
       aggr-ref 8-25(70.1)
```

```
(101) simple-def-gen(gen,eva) =
    test-simple-def(s-eva(gen),eva) -->
    μ0(<s-eva:eva>,<s-mi:m-1(gen)>,<s-pp:s-pp(gen)>)
    T -- error
```
Note: This axiom states that transition from a value representation to a value and back to a value representation leaves the given value representation unchanged. (The stronger consequence mentioned above (cf. (15)) concerns the opposite transition: from a value via a representation back to the value).

\( vr \in vr\text{-set}(eva) \Rightarrow \text{rep}(eva, \text{val}(eva, vr)) = vr \)

Note: This axiom expresses the fact that unaligned strings are represented linearly in storage.

\( (23.1) \quad vr \in vr\text{-set}(\text{str-eva}(base, j)) \Rightarrow \)
\( \forall j \in \text{val}(\text{str-eva}(base, j), vr) = \text{CONC} \sum_{i=1}^{j} \text{val}(\text{str-eva}(base, 1), \text{str-part}(base, i, 1)(vr)) \)

for: \( (\text{is-BIT} \lor \text{is-CHAR})(base) \)

Ref.: str-eva 7-14(45)
str-part 7-13(39)

Note: This axiom expresses the fact that unaligned strings are represented linearly in storage.

\( (23.2) \quad \text{is-pic\text{-}s\text{-}da}(eva) \land \text{is-UNAL\text{-}s\text{-}dens}(eva) \land vr \in vr\text{-set}(eva) \Rightarrow \)
\( \forall eva, vr \in \text{val}(eva, vr) = \text{val}(\text{str-eva}(\text{str-base\text{-}s\text{-}da}(eva), \text{str-length\text{-}s\text{-}da}(eva)), vr) \)

Ref.: str-eva 7-14(45)
str-base 9-41(163)
str-length 9-41(164)

Note: This axiom is an analogue of axiom (23.1) for the case of unaligned pictures.

\( (26) \quad \text{value}(eva, vr) = \)
\( \text{is-pic\text{-}s\text{-}da}(eva) \Rightarrow \text{pic-val}(s\text{-}da(eva), \text{val}(eva, vr)) \)
\( T \Rightarrow \text{val}(eva, vr) \)

Ref.: pic-val 9-62(240)

Before representing a value, the instruction test-rep(eva,v) tests whether SIZE or STRZ conditions have to be raised. If it is known that no condition can occur, then the function rep(eva,v) can be used directly.
(27) \texttt{test-real}(eva,v) =
\begin{align*}
\text{(is-arithm} & \& \text{is-CPLX-s-mode)} \& \text{s-da(eva)} \& \\
\text{(is-size-cond(eda-real,real(v))} \& \text{is-size-cond(eda-real,imag(v))}) \& \\
\text{is-REAL-s-mode} \& \text{s-da(eva)} \& \text{is-size-cond(s-da(eva)}, \text{real(v)} \\
\text{call-cond}(SIZE) \\
\text{is-string} \& \text{s-da(eva)} \& \text{is-str-size-cond(s-da(eva),v)} \rightarrow \\
\text{pass-rep}(eva,\text{adjust-val(s-da(eva),v)}); \\
\text{call-cond}(STR) \\
\text{is-pic}s-da(eva) \rightarrow \text{rep-pic}(eva,v) \\
T \rightarrow \text{PASS:rep}(eva,\text{adjust-val(s-da(eva),v)}) \\
\end{align*}

where:
\begin{align*}
\text{eda-real} = \mu(s-da(eva); <s-mode:REAL>)
\end{align*}

Ref.: \texttt{call-cond} 10-18(54)
\texttt{rep-pic} 9-51(204)

(28) \text{is-size-cond}(eda,v) =
\begin{align*}
\text{is-FIX-s-scale}(eda) \rightarrow \abs{v} \times \text{base-eda,} + q-eda) \geq \text{base-eda,} + p-eda \\
\text{is-FIT-s-scale}(eda) \rightarrow v \neq 0 \& -(\text{min-flt-eda,} \leq \abs{v} \leq \text{max-flt-eda,})
\end{align*}

for: (is-prop-arithm \& \text{is-REAL-s-mode}(eda) \& is-real-val(v)

(29) \text{is-str-size-cond}(eda,v) =
\begin{align*}
s-length(eda) < \text{length}(v)
\end{align*}

for: is-string(eda) \& (is-char-val-list \& is-bit-val-list)(v)

(30) \text{adjust-val}(eda,v) =
\begin{align*}
\text{is-REAL-s-mode}(eda) \rightarrow \text{real}(v) \\
\text{is-string}(eda) \rightarrow \text{adjust-string(s-base(eda),result-length(eda,v),v)} \\
T \rightarrow v
\end{align*}

(31) \text{result-length}(eda,v) =
\begin{align*}
\text{is-0-s-varying(eda)} \rightarrow s-length(eda) \\
T \rightarrow \text{length}(v)
\end{align*}

for: is-string(eda) \& (is-char-val-list \& is-bit-val-list)(v)

8 9. DATA, OPERATIONS AND CONVERSIONS
9. OPERANDS

The first formula defines an operand as an object that has two parts, an evaluated aggregate attribute and a value representation. The rest is an auxiliary instruction for producing operands from their parts, two functions for transforming an operand into its value or conversely, and two functions and two predicates producing and characterizing integer operands.

(34) \[ \text{is-op} = \]
\[ (\langle s\text{-eva} : \text{is-ev}\rangle, \]
\[ \langle s\text{-vr} : \text{is-vr}\rangle) \]

Ref.: is-vr 3-15(32)

Note: Actually only operands will be produced whose eda-parts satisfy the predicate is-correct-eda.

(35) \[ \text{mk-op}(eva, vr) = \]
\[ \text{PASS}: \mu_0(\langle s\text{-eva} : e\rangle, \langle s\text{-vr} : v\rangle) \]

(36) \[ \text{op-val}(op) = \]
\[ \text{value}(eva-op, vr-op) \]
for: \( \text{is-area}\star s\text{-da}(eva-op) \)

(37) \[ \text{op-da}(op) = \]
\[ s\text{-da}\star s\text{-eva}(op) \]

(37.1) \[ \text{val-op}(eda, v) = \]
\[ \mu_0(\langle s\text{-eva} : \text{aggr-scalar}(eda)\rangle, \langle s\text{-vr} : \text{rep}(\text{aggr-scalar}(eda), \text{adjust-val}(eda, v))\rangle) \]
for: \( \text{is-pic}(eda) \)
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

31 October 1969

Ref.: agrv-scalar B-24(64)

Note: Since rep and not test-rep appears, the function can only be used in cases where no SIZ or STRZ conditions can arise.

(39) intg-op(i) =
    val-op(INT3-EDA,i)
(40) bintg-op(i) =
    val-op(BINT3-EDA,i)
(41) is-intg-op(x) =
    (is-op & is-intg•s-da)(x)
    ref.: is-intg 9-11(51)
(42) is-bintg-op(x) =
    (is-op & is-bintg•s-da)(x)

9.1.5 OPERATORS

An operator is either an infix operator, or a prefix operator, or the object C-EXP introduced by the instruction EVAL-EXPR to treat a special case of exponentiation.

(43) is-operator =
    is-infix-opr v is-prefix-opr v is-C-EXP
(44) is-aritma-opr =
    is-ADD v is-SUBTR v is-MULT v is-DIV v is-EXP v is-C-EXP v is-PLUS v is-MINUS
(45) is-comp-opr =
    is-GE v is-GE v is-LE v is-LT v is-NE
(46) is-bit-opr =
    is-AND v is-OR v is-NOT

9.2 AUXILIARY DEFINITIONS ON DATA ATTRIBUTES

The definitions given in this section are used in chapter 9 and in other chapters. In 9.2.1, integer data attributes, functions producing integer or string data attributes of given precision, scale or length, and a few incomplete

10 9. DATA, OPERATIONS AND CONVERSIONS
(130) \texttt{num-bit-conv}(eda,v) = \\
\texttt{lgth-eda}_1 \leq 0 \rightarrow \text{PASS}:<> \\
T \rightarrow \text{bintg-bit-conv}(\text{lgth-eda}_1,v) \\
\text{vr} : \text{test-rep}(\text{bintg-eda}(\text{lgth-eda}_1),v)

where: \\
\text{lgth-eda}_1 = \text{bit-length} (\text{mk-arithm}(eda))

for: \text{is-num-type}(eda) \& \text{is-num-val}(v)

Ref.: \begin{align*}
\text{test-rep} & 9-9(27) \\
\text{bintg-eda} & 9-11(48) \\
\text{is-num-type} & 9-21(91) \\
\text{is-num-val} & 9-3(3)
\end{align*}

(131) \texttt{bintg-bit-conv}(n,vr) = \\
\text{PASS}: \text{num-bit-list}(n,\text{abs-value}(\text{bintg-eda}(n),vr))

Ref.: \begin{align*}
\text{value} & 9-7(26) \\
\text{bintg-eda} & 9-11(48)
\end{align*}

(132) \texttt{num-bit-list}(n,v) = \\
\texttt{n} = 0 \rightarrow <> \\
T \rightarrow \text{num-bit-list}(n-1,\text{trunc}(v/2))^<\text{single-num-bit(modulo}(v,2))>

for: \text{is-real-val}(v) \& n \geq 0

Ref.: \begin{align*}
\text{is-real-val} & 9-3(4)
\end{align*}

(133) \texttt{single-num-bit}(v) = \\
(v = 0 \rightarrow \text{0-BIT,} \\
\text{v} = 1 \rightarrow \text{1-BIT})

for: v = 0 \lor v = 1

9.5.1.3 Character to numeric conversion

The instruction \texttt{char-num-conv}(v) is defined in chapter 11, because it uses methods for parsing strings that are developed there (cf. 11.6.1).

9.5.1.4 Character to bit conversion

The character string is parsed from left to right. If a character is found that cannot be converted then the \texttt{CONVERSION} condition is raised. On return from the on-unit, conversion is retried with the corrected string.
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

(133.1) \text{char-bit-conv}(eda-tg,v) =
\begin{align*}
\text{lgth-tg}_1 & = \min(\text{length}(v), \text{str-length}(eda-tg)) \\
\text{i}_0 & = \{i | \{1 \leq i \leq \text{lgth-tg}_1 \land \text{is-0-CHAR} \lor \text{is-1-CHAR}\} \cdot \text{elem}(i,v) \land \\
& \{j | 1 \leq j < i \land \text{is-0-CHAR} \lor \text{is-1-CHAR}\} \cdot \text{elem}(j,v)\}\}
\end{align*}

\text{T --}
\begin{align*}
\text{char-bit-conv}(eda-tg,\text{corr-v}); \\
\text{corr-v:call-conv-const}(v,i_0)
\end{align*}

where:
\begin{align*}
\text{lgth-tg}_1 & = \min(\text{length}(v), \text{str-length}(eda-tg)) \\
\text{i}_0 & = \{i | \{1 \leq i \leq \text{lgth-tg}_1 \land \text{is-0-CHAR} \lor \text{is-1-CHAR}\} \cdot \text{elem}(i,v) \land \\
& \{j | 1 \leq j < i \land \text{is-0-CHAR} \lor \text{is-1-CHAR}\} \cdot \text{elem}(j,v)\}\}
\end{align*}

for: \(\text{is-string} \land \text{is-BIT}\cdot s\cdot base \cdot (eda-tg) \land \text{is-char-val-list}(v)\)

Ref.: call-conv-const 10-24(70) is-char-val 9-3(6)

(135) \text{single-char-bit}(v) =
\begin{align*}
\text{is-0-CHAR}(v) & \rightarrow \text{0-BIT}, \\
\text{is-1-CHAR}(v) & \rightarrow \text{1-BIT}
\end{align*}

for: \(\text{is-0-CHAR} \land \text{is-1-CHAR} \cdot (v)\)

9.5.1.5 Bit to numeric conversion

(136) \text{bit-num-conv}(v) =
\begin{align*}
\text{length}(v) \sum_{i=1}^{\text{single-bit-num}(\text{elem}(i,v))} \cdot 2 \cdot (\text{length}(v) - i)
\end{align*}

for: \text{is-bit-val-list}(v)

Ref.: is-bit-val 9-4(11)

(137) \text{single-bit-num}(v) =
\begin{align*}
\text{is-0-BIT}(v) & \rightarrow \text{0}, \\
\text{is-1-BIT}(v) & \rightarrow \text{1}
\end{align*}

for: \text{is-bit-val}(v)
Abstract Syntax and Interpretation of PL/I

Ref.: is-bit-val 9-4(11)
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

(3) \text{is-corr-check}(\text{ref}, \text{at}) = \text{is-}2\text{c-p-tr}(\text{ref}) \& \text{is-c-}\text{s-s}(\text{ref}) \& \text{is-c-}\text{s-ap}(\text{ref}) \& \text{is-c-}\text{p-v-a}\text{r}(\text{attr}_0) \& \text{is-c-}\text{p-s-scope}(\text{attr}_0) \& \text{is-c-}\text{e-}\text{n-}\text{t-}\text{y-}\text{c-}\text{on-}\text{st}(\text{attr}_0) \& \text{is-c-}\text{l-a-b-e-l-}\text{c-}\text{on-}\text{st}(\text{attr}_0))

where:
\text{attr}_0 = \text{s-n}(\text{ref}) (\text{at})

for: \text{is-c-ref}(\text{ref})

(3.1) \text{eval-pref-cond}(\text{condl}, \text{at}) = \text{is-c-}(\text{condl}) \rightarrow \text{<>}
\text{is-c-}\text{c-c}(\text{condl}) \rightarrow \text{<>}
\text{is-c-}\text{e}\text{-r-}\text{l}\text{h}(\text{condl}) \rightarrow \text{<>}
\text{exp-r-l}(\text{s-c-r-}\text{r}\text{-}\text{r}-\text{l}\text{h}\text{-}\text{h}(\text{condl}), \text{at}) \text{eval-pref-cond}(\text{t-a}\text{l}(\text{condl}), \text{at})
\text{t} \rightarrow \text{h-e-a-d}(\text{condl}) \text{eval-pref-cond}(\text{t-a}\text{l}(\text{condl}), \text{at})

for: \text{is-c-}\text{p-}\text{c-c-}\text{d}\text{-}\text{l}(\text{condl})

Ref.: \text{exp-r-l} 10-22(66)

(5) \text{merge-pref}(\text{bpp}, \text{cdl-1, cdl-2, at}) = \mu(\text{bpp}, \text{cdl})\text{i} \cdot \{<\text{cond-sel}(\text{elem}(i, \text{cdl-1}), \text{at}): <\text{cond-sel}(\text{elem}(i, \text{cdl-2}), \text{at}): \text{is-c-}\text{c-}\text{d}\text{-}\text{p}\text{-}\text{t}(\text{bpp}) \text{for: is-c}\text{-}\text{p}\text{-}\text{c-c-}\text{d}(\text{bpp})

Ref.: \text{cond-sel} 3-20(73.1)
\text{is-c}\text{-}\text{p}\text{-}\text{c-c-}\text{d}\text{-}\text{p}\text{-}\text{t}(\text{bpp}) 3-21(79.1)

10.1.1.2 Statement prefixes

During the interpretation of PL/I statements the enabling status as defined for the block can be modified by explicit statement condition prefixes.

(6) \text{up-c-t-c-e}(\text{cpp}, \text{cpp}, \text{at}) = \gamma (\text{cpp}) \{1 \leq i \leq \text{length}(\text{cpp}) \& \text{is-c-}\text{c-}\text{d}\text{-}\text{p}(\text{cpp}) \& \text{is-c-}\text{c-}\text{d}\text{-}\text{p}(\text{cpp}) \& \text{is-c-}\text{c-}\text{d}\text{-}\text{p}(\text{cpp}) \rightarrow \text{<>}
\text{s-c-c-} \mu(\text{cpp}, <\text{cpp}: \text{merge-pref}(\text{cpp}, \text{cpp}, \text{cpp}, \text{at}))
\text{t} \rightarrow \text{e-x-i-l-e-o-n}

10.1.2 ENABLING AND DISABLING OF ATTENTIONS

The enabling and disabling of attentions is controlled for each task by the enable and disable statements, respectively. Information necessary for all tasks is stored in the global directory and under the specific attention identification,
whereas the attentions a task has enabled or associated, with or without the event option, are kept in the task local state component $EN$.

**Metavariables**

cdl | is-eattn-cond-list | evaluated attention condition list
cond | is-eattn-cond | evaluated attention condition

### 10.1.2.1 Enable statement

The interpretation distinguishes between enabling with or without event option and thereby makes entries into the different components of $EN$. The attention directory $AN$ is updated with every enable statement.

When an attempt has been made to enable all the attentions of the enable statement, then the inspection of the $s$-wait-list component of $EN$ decides whether the task has to wait or may continue. Events for which all attentions are successfully enabled are immediately deleted.

When the enabling mode is changed to asynchronous by the enable statement and the info-list of the relevant attention in $AN$ is not empty, an asynchronous interrupt is immediately executed.

The attentions as described in the abstract syntax of an enable or disable statement are evaluated and satisfy the predicate is-eattn-cond. The identification part of this evaluated condition is now comparable with the identification part of the incoming attention which describes the attention occurrence outside the PL/I machine.

\[(7) \quad \text{int-enable-st}(t) = \]
\[
\quad \text{int-enable-st}(s\text{-list}(t))
\]

for: is-enable-st(t)
(7.1) \[\text{int-enable-1}(\text{enable-list}) =\]

\[\begin{align*}
\text{is-<>}(\text{enable-list}) \& \text{is-<>}(\text{s-event-head}(\text{enable-list})) \rightarrow \\
\text{int-enable-1}(\text{tail}(\text{enable-list})) ; \\
\text{int-enable-2}(\text{tn}, \text{head}(\text{enable-list}), \text{cdl}) ; \\
\text{attach-att-event}(\text{gen}, \text{tn}) ; \\
\text{cdl}: \text{pass-<att-list}(\text{idr}_1, \text{DN}, \text{EV}) ; \\
\text{gen}: \text{eval-ref-<gen}(\text{s-event-head}(\text{enable-list})) ; \\
\text{tn}: \text{un-name}
\end{align*}\]

\[\text{is-<>}(\text{enable-list}) \rightarrow\]

\[\begin{align*}
\text{int-enable-1}(\text{tail}(\text{enable-list})) ; \\
\text{int-enable-2}(\text{0}, \text{head}(\text{enable-list}), \text{cdl}) ; \\
\text{cdl}: \text{pass-<att-list}(\text{idr}_1, \text{DN}, \text{EV}) \\
\text{is-<>}(\text{enable-list}) \& \text{is-<>}(\text{s-wait-list}(\text{tn})) \rightarrow \text{enab-wait}
\end{align*}\]

T \rightarrow \text{null}

where:
\[\text{idr}_1 = \text{s-att-list}\text{-s-cond-head}(\text{enable-list})\]

for: \text{is-enable-list}(\text{enable-list})

Ref.: \text{eval-ref-gen} 8-28(82)
\text{un-name} 3-10(20)

(7.2) \[\text{int-enable-2}(\text{tn}, \text{enable}, \text{cdl}) =\]

\[\begin{align*}
\text{is-<>}(\text{cdl}) \& (\text{is-<>}(\text{attn}_1) \lor \text{s-task}(\text{attn}_1) = \text{TN}) \rightarrow \\
\text{int-enable-2}(\text{tn}, \text{enable}, \text{tail}(\text{cdl})) ; \\
\text{enable}(\text{enable}, \text{head}(\text{cdl})) \\
\text{is-<>}(\text{cdl}) \rightarrow \\
\text{int-enable-2}(\text{tn}, \text{enable}, \text{tail}(\text{cdl})) ; \\
\text{associate}(\text{tn}, \text{enable}, \text{head}(\text{cdl})) \\
\text{is-<>}(\text{cdl}) \& \text{is-<>}(\text{tn}) \& (\text{is-<>}(\text{tn}(\text{tn})) \lor \text{is-<>}(\text{tn}(\text{tn}))) \rightarrow \\
\text{delete-task-event}(\text{tn}, \text{NORMAL})
\end{align*}\]

T \rightarrow \text{null}

where:
\[\text{attn}_1 = \text{s-ident}(\text{head}(\text{cdl})) (\text{AN})\]

for: \text{is-n}(\text{tn}) \lor \text{is-<>}(\text{tn}), \text{is-enable}(\text{enable})

Ref.: \text{delete-task-event} 5-8(28)
10. ATTENTIONS AND CONDITIONS

(10) \[\text{enable}(\text{enable}, \text{cond}) =\]

\[\text{is-q}(\text{attn}_1) \rightarrow\]

\[\text{s-an}: \mu(\text{AN};<\text{ident}_1: \mu_0(\langle \text{s-info}: <\text{cond}>\rangle, \langle \text{s-task}: \text{TN}\rangle, \langle \text{s-spec}: \text{s-spec}(\text{enable})\rangle, \langle \text{s-assoc}: \{\}\rangle)\rangle\]

\[\text{s-en}: \mu(\text{EN};<\text{enab-list}: \text{enab-list}(\text{EN}) \times<\text{cond}>\rangle)\]

\[\text{s-spec}(\text{enable}) = \text{s-spec}(\text{attn}_1) \rightarrow \text{null}\]

\[\text{is-ACC} \times \text{s-spec}(\text{attn}_1) \& \text{is-ASYN} \times \text{s-spec}(\text{enable}) \& \neg \text{is-}<> \times \text{s-info}(\text{attn}_1) \rightarrow\]

\[\text{s-an}: \mu(\text{AN};<\text{s-spec-ident}_1: \text{s-spec}(\text{enable})\rangle)\]

\[\text{s-en}: \mu(\text{EN};<\text{enab-list}: \text{enab-list}(\text{EN}) \times<\text{cond}>\rangle, \langle \text{tn}: \text{tn}(\text{EN}) \times<\text{cond}>\rangle)\]

\[\text{T} \rightarrow \text{s-en}: \mu(\text{EN};<\text{s-spec-ident}_1: \text{s-spec}(\text{enable})\rangle)\]

where:

\[\text{ident}_1 = \text{s-ident}(\text{cond})\]

\[\text{attn}_1 = \text{id}_1(\text{AN})\]

for: \text{is-enable}(\text{enable})

Ref.: prep-attn-1 10-14(39)

(11) \[\text{associate}(\text{tn}, \text{enable}, \text{cond}) =\]

\[\neg \text{is-q}(\text{tn}) \& (\forall \text{tn-1}(\text{tn-1} \in \text{s-assoc}(\text{attn}_1) \& \text{tn-1} \neq \text{TN}) \rightarrow\]

\[\text{s-an}: \mu(\text{AN};<\text{ident}_1: \mu_0(\langle \text{s-assoc}: \text{s-assoc}(\text{attn}_1) \cup \{\text{TN}\}\rangle)\rangle\]

\[\text{s-en}: \mu(\text{EN};<\text{assoc-list}: \text{assoc-list}(\text{EN}) \times<\text{cond}>\rangle, \langle \text{tn}: \text{tn}(\text{EN}) \times<\text{cond}>\rangle)\]

\[(\forall \text{tn-1}(\text{tn-1} \in \text{s-assoc}(\text{attn}_1) \& \text{tn-1} \neq \text{TN}) \rightarrow\]

\[\text{s-en}: \mu(\text{EN};<\text{assoc-list}: \text{assoc-list}(\text{EN}) \times<\text{cond}>\rangle, \langle \text{tn}: \text{tn}(\text{EN}) \times<\text{cond}>\rangle)\]

\[\neg \text{is-q}(\text{tn}) \& (\exists \text{tn-1}(\text{tn-1} \in \text{s-assoc}(\text{attn}_1) \& \text{tn-1} = \text{TN}) \rightarrow\]

\[\text{s-en}: \mu(\text{EN};<\text{assoc-list}: \text{assoc-list}(\text{EN}) \times<\text{cond}>\rangle, \langle \text{tn}: \text{tn}(\text{EN}) \times<\text{cond}>\rangle)\]

\[\text{T} \rightarrow \text{s-en}: \mu(\text{EN};<\text{assoc-list}: \text{assoc-list}(\text{EN}) \times<\text{cond}>\rangle)\]

where:

\[\text{ident}_1 = \text{s-ident}(\text{cond})\]

\[\text{attn}_1 = \text{id}_1(\text{AN})\]

\[\text{cond}_1 = \mu_0(\langle \text{s-ident}: \text{s-ident}(\text{cond})\rangle, \langle \text{s-spec}: \text{s-spec}(\text{enable})\rangle, \langle \text{tn}: \{\text{tn}\}\rangle)\]

for: \text{is-q} \lor \text{is-q}(\text{tn}), \text{is-enable}(\text{enable})

6
(11.1) \[ \text{insert-list}(\text{list}, \text{cond}, \text{opt}) = \]
\[ s\text{-ident}(\text{head}(\text{list})) = s\text{-ident}(\text{cond}) \rightarrow \]
\[ \text{merge}\text{-cond}(\text{cond}, \text{head}(\text{list}), \text{opt})\text{-tail}(\text{list}) \]
\[ T \rightarrow \text{insert-1}(\text{head}(\text{list}), \text{tail}(\text{list}), \text{cond}, \text{opt}) \]
\[ \text{for:} \text{is-opt} (\text{opt}) \]

(11.2) \[ \text{insert-1}(\text{list-1}, \text{list-2}, \text{cond}, \text{opt}) = \]
\[ \neg \text{is-<>}(\text{list-2}) \in s\text{-ident}(\text{head}(\text{list-2})) = s\text{-ident}(\text{cond}) \rightarrow \]
\[ \text{list-1}\text{-merge}\text{-cond}(\text{cond}, \text{head}(\text{list-2}), \text{opt})\text{-tail}(\text{list-2}) \]
\[ \neg \text{is-<>}(\text{list-2}) \in \text{insert-1}(\text{list-1}\text{-head}(\text{list-2}), \text{tail}(\text{list-2}), \text{cond}, \text{opt}) \]
\[ T \rightarrow \text{list-1} \]
\[ \text{for:} \text{is-opt} (\text{opt}) \]

(14) \[ \text{merge}\text{-cond}(\text{cond-1}, \text{cond-2}, \text{opt}) = \]
\[ \text{is-<>}(\text{opt}) \rightarrow \langle \rangle \]
\[ \text{is-<>s-tn}(\text{cond-2}) \rightarrow \langle \text{cond-1} \rangle \]
\[ \emptyset \rightarrow \]
\[ \langle \mu_0(\langle s\text{-ident:s-ident}(\text{cond-1}), s\text{-spec:s-spec}(\text{cond-1}), s\text{-tn:s-tn}(\text{cond-1}) \cup s\text{-tn}(\text{cond-2}) \rangle) \rangle \]
\[ \text{for:} \text{is-opt} (\text{opt}) \]

(15) \[ \text{eval}\text{-attn-list}(\text{idrl}, \text{dn}, \text{ev}) = \]
\[ \text{is-<>}(\text{idrl}) \rightarrow \langle \rangle \]
\[ T \rightarrow \langle \text{eval}\text{-attn}(\text{head}(\text{idrl}), \text{dn}, \text{ev}) \rangle \text{-eval}\text{-attn-list}(\text{tail}(\text{idrl}), \text{dn}, \text{ev}) \]
\[ \text{for:} \text{is-id-ref-list}(\text{idrl}) \]

(16) \[ \text{eval}\text{-attn}(\text{idr}, \text{dn}, \text{ev}) = \]
\[ \mu_0(\langle s\text{-ident:mk}\text{-ident}(s\text{-id}(\text{idr}), s\text{-n}(\text{idr})(\text{dn})(\text{ev})) \rangle) \]
\[ \text{for:} \text{is-id-ref}(\text{idr}) \]

(17) \[ \text{mk}\text{-ident}(\text{id}, \text{ea}) = \]
\[ \text{for:} \text{is-id}(\text{id}), \text{is-ea}(\text{ea}) \]

cont'd
Ref.: is-ea 11-22(58)

Note: Implementation defined selector value, identifying an attention.

(18) \((\text{vid, ea}) \text{(is-ident.mk-ident(id, ea)})\)

Ref.: is-ident 1-7(5.1)

(19) \(\text{attach-attn-event}(\text{gen, tn})\) =
\[\text{is-active-event}(\text{gen}) \rightarrow \text{call-cond}(\text{ERROR})\]
\[-\text{is-active}(\text{gen, PA}) \rightarrow\]
\[s => \text{el-ass}(\text{ev-vr}_0, \text{s-pp}(\text{gen}), s)\]
\[s => \mu(\text{PA}: <\text{tn}: \mu_0 (\langle s \text{-ev}, s \text{-te}: \text{gen} \rangle)>)\]
\[T \rightarrow \text{error}\]

where:
\[\text{ev-vr}_0 = \text{rep}(\text{s-eva}(\text{gen}), \mu_0 (\langle s \text{-compl:0-BIT}, s \text{-status:0} \rangle))\]

for: is-gen(\text{gen}, is-tn(\text{tn}))

Ref.: is-active-event 5-5(11)
call-cond 10-18(54)
is-active 5-5(12)
el-ass 3-16(48)
rep 9-6(19)
is-gen 3-14(30)
is-tn 3-4(6)

(20) \(\text{enab-wait} = \)
\[\text{is-<>s-wait-list}(EY) \rightarrow \text{null}\]
\[T \rightarrow \]
\[\text{enab-wait; get-wait-state-3}\]

(21) \(\text{set-wait-state-3} = \)
\[\text{is-<>s-wait-list}(EY) \rightarrow \text{null}\]
\[T \rightarrow \text{ste}: \mu(\text{FP}: <\text{s-wait: *})\]

10.1.2.2 disable statement

The disable statement disables or dissociates the attentions from the task and possibly enables them in another task; thereby it may delete events or execute asynchronous interrupts. Again the attentions in the disable statement are evaluated as in the enable statement.
(21.1) \[ \text{int-disable-st}(t) = \]
\[ \text{int-disable-1}(cdl); \]
\[ \text{cdl:pass-eval-attn-list}(s\text{-attn-list}\cdot s\text{-cond}(t),DN,EX) \]
\[ \text{for:is-enable-st}(t) \]
(23) \texttt{int-disable-1(cdl)} = \\
is-\Rightarrow(cdl) \rightarrow \texttt{null} \\
s-task(attn_1) = \texttt{NW} \rightarrow \\
\hspace{1em} \texttt{int-disable-1(tail(cdl))}; \\
\hspace{1em} \texttt{disable}(head(cdl)) \\
(\exists tn) (tn \in s-assoc(attn_1) \& tn = \texttt{NW}) \rightarrow \\
\hspace{1em} \texttt{int-disable-1(tail(cdl))}; \\
\hspace{1em} \texttt{disassociate}(head(cdl)) \\
\texttt{T} \rightarrow \texttt{int-disable-1(tail(cdl))} \\

\text{where:} \\
\hspace{1em} attn_1 = s-ident(head(cdl)) (\texttt{AN}) \\

(23.1) \texttt{disable}(cond) = \\
is-\{} \& s-assoc(attn_1) \rightarrow \\
\hspace{1em} s-an: (\texttt{AN}; attn_1) \\
\hspace{2em} s-an: \mu (\texttt{EN}; \langle s-enab-list:insert-list(s-enab-list(\texttt{EN}), \texttt{cond}, \ast ) \rangle) \\
\hspace{1em} \texttt{is-\Rightarrow s-info(attn_1) \& s-spec(\texttt{cond}) = \texttt{ASYN} \rightarrow} \\
\hspace{2em} s-an: \texttt{an}_1 \\
\hspace{3em} s-en: \mu (\texttt{EN}; \langle s-enab-list:insert-list(s-enab-list(\texttt{EN}), \texttt{cond}, \ast ) \rangle) \\
\hspace{3em} s-en: \texttt{activate-tasks}(\mu (\texttt{EN}; \langle s-en\texttt{tn}_1: \texttt{en}_1 \rangle)) \\
\hspace{2em} s-s: \texttt{null}; \\
\hspace{3em} \langle \texttt{delete-task-event}(\texttt{tn}, \texttt{NORMAL}) \mid \texttt{tn} \in s\texttt{-tn(\texttt{cond})} \& \texttt{is-\Rightarrow TN(\texttt{en}_1) \rangle} \rangle \\
\texttt{T} \rightarrow \\
\hspace{1em} s-an: \texttt{an}_1 \\
\hspace{2em} s-en: \mu (\texttt{EN}; \langle s-enab-list:insert-list(s-enab-list(\texttt{EN}), \texttt{cond}, \ast ) \rangle) \\
\hspace{3em} s-en: \texttt{activate-tasks}(\mu (\texttt{EN}; \langle s-en\texttt{tn}_1: \texttt{en}_1 \rangle)) \\
\hspace{2em} s-s: \texttt{null}; \\
\hspace{3em} \langle \texttt{delete-task-event}(\texttt{tn}, \texttt{NORMAL}) \mid \texttt{tn} \in s\texttt{-tn(\texttt{cond})} \& \texttt{is-\Rightarrow TN(\texttt{en}_1) \rangle} \rangle \\

\text{where:} \\
\hspace{1em} \texttt{ident}_1 = s-ident(\texttt{cond}) \\
\hspace{1em} attn_1 = \texttt{ident}_1 (\texttt{AN}) \\
\hspace{2em} \hspace{1em} \texttt{tn}_n = \texttt{head*order-set}(s\texttt{-assoc(attn}_1, \texttt{DISABLE}) \\
\hspace{2em} \hspace{1em} \texttt{cond}_n = (\texttt{1x})(\texttt{3i})(x = \texttt{elem}(i, s\texttt{-assoc-list(s-en*tn}_1(\texttt{PA}))) \& \\
\hspace{5em} \texttt{1} \leq i \leq \texttt{length(s\texttt{-assoc-list(s-en*tn}_1(\texttt{PA}))))} \& \\
\hspace{4.5em} \texttt{s-ident(x) = s-ident(\texttt{cond})} \\
\hspace{2em} \hspace{1em} \texttt{task}_1 = \mu (\texttt{tn}_1(\texttt{PA}); \langle s\texttt{-en}\texttt{en}_1 \rangle) \\
\hspace{2em} \hspace{1em} \texttt{en}_1 = s\texttt{-en(task}_1) \\
\hspace{2em} \hspace{1em} \texttt{en}_2 = s\texttt{-en(task}_1) \\
\hspace{2em} \hspace{1em} \texttt{en}_2 = \mu (\texttt{en}_2; \langle s\texttt{-enab-list:insert-list(s-enab-list(\texttt{en}_2), \texttt{cond}, \ast ) \rangle,} \\
\hspace{5em} \langle s\texttt{-wait-list:insert-list(s-wait-list(\texttt{en}_2), \texttt{cond}, \ast ) \rangle,} \\
\hspace{5em} \langle \texttt{tn}\texttt{-insert-list(\texttt{en}_2, \texttt{cond}, \ast ) \rangle \mid \texttt{tn} \in s\texttt{-tn(\texttt{cond})}) \rangle \\
\hspace{2em} \hspace{1em} \texttt{an}_1 = \mu (\texttt{AN}; \langle s\texttt{-spec*ident}_1; s\texttt{-spec(\texttt{cond})}, \langle s\texttt{-task*ident}_1; \texttt{tn}_1 \rangle,} \\
\hspace{5em} \langle s\texttt{-assoc*ident}_1; s\texttt{-assoc(attn}_1) \rightarrow \{ \texttt{tn}_1 \} \rangle) \\
\hspace{1em} \text{Ref.:} \\
\hspace{1em} \texttt{activate-tasks} 5-12(42) \\
\hspace{1em} \texttt{prep-attn-1} 10-14(39) \\
\hspace{1em} \texttt{delete-task-event} 5-8(28) \\
\hspace{1em} \texttt{order-set} 11-41(111)
10.2 CONDITION ACTION

Condition actions are the actions which have to be performed if in the interpretation of a PL/I program a condition situation occurs, the condition is raised, the condition is enabled and the normal flow of control is interrupted as specified for the specific raising. The action to be executed can either be the standard system action or the action specified by the execution of on and revert statements.

10.2.1 STANDARD SYSTEM ACTION FOR CONDITIONS

If the interpretation of a program specifies that a condition action is to be executed and no on-unit has been established for the condition in the current CS, the system action defined by syst-cond-exec is executed.

(25.1) syst-cond-exec (cond, cbif) =

is-ONL (cond) \lor is-STRG (cond) \lor is-STRZ (cond) \lor is-NAME•s-cond (cond) \lor
is-progr-named-cond (cond) \rightarrow
comment (cond, cbif)

(is-ENDP•s-cond (cond) \land is-SIGNAL•s-type (cbif)) \lor is-E0V•s-cond (cond) \lor
is-E0V•s-cond (cond) \lor is-PEND•s-cond (cond) \lor is-FINISH (cond) \lor
is-eattn-cond (cond) \rightarrow

pull

is-ENDP•s-cond (cond) \rightarrow syst-endpage-exec (s-f (cond))

is-CONV (cond) \lor is-io-cond•s-cond (cond) \lor is-FOFL (cond) \lor is-OFL (cond) \lor
is-SIZE (cond) \lor is-SDIV (cond) \lor is-SUBRG (cond) \lor is-AREA (cond) \rightarrow
call-cond-l (ERROR, cbif) ;

comment (cond, cbif)

is-check (cond) \rightarrow syst-check-exec (cond)

is-ERROR (cond) \rightarrow syst-error-exec

where:

cbif = \mu (cbif; <s-cond; cond>)

Ref.: is-eattn-cond 3-19(72)
syst-endpage-exec 11-123(332)
call-cond-l 10-19(55)
is-check 3-21(77)
syst-check-exec 11-123(331)
(27) \[
\text{comment}\text{(cond, cbif)} = \\
\text{s-\(\text{\#}M: s-\text{comment}\text{: s-comment\text{(}}M\text{)} \cdot \text{comment-f\{cond, cbif, t\}\}}
\]
(41) \[ \text{call-attr-cond}(\text{cond}) = \]
\[ \text{freq}(b); \]
\[ \text{change-enab-1}(\text{cond}); \]
\[ \text{call-cond-1}(\text{cond}, \text{cbif}); \]
\[ \text{change-enab}(\text{cond}); \]
\[ \text{cbif}\text{page-attr-chif}(\text{gen}, \text{cond}, \text{cbif}); \]
\[ \text{install-info}(\text{gen}, \text{cond}); \]
\[ \text{gen}\text{allocate}(\text{b}, \text{eva}_0, \text{DUMMY}); \]
\[ \text{b}\text{un-name} \]

where:
\[ \text{eva}_0 = \text{s-struct}\text{head}\text{s-info}(\text{s-ident}(\text{cond})(\text{AN})) \]

Ref.: \[ \text{free \ 7-18.1(58)} \]
\[ \text{call-cond-1 \ 10-19(55)} \]
\[ \text{allocate \ 7-11(20)} \]
\[ \text{un-name \ 3-10(20)} \]

(42) \[ \text{install-info}(\text{gen}, \text{cond}) = \]
\[ \text{s-an}: (\text{AN}; <\text{s-info}\text{ident}_1\text{tail}\text{s-info}(\text{attn}_1)> ) \]
\[ \text{s-c assignable}(\text{gen}, \text{opo}) \]

where:
\[ \text{ident}_1 = \text{s-ident}(\text{cond}) \]
\[ \text{attn}_1 = \text{ident}_1(\text{AN}) \]
\[ \text{opo}_0 = \text{s-op}\text{head}\text{s-info}(\text{attn}_1) \]

Ref.: \[ \text{assign \ 8-10(23)} \]

(43) \[ \text{change-enab}(\text{cond}) = \]
\[ \text{is-ASYN}\text{s-spec(\text{attn}_1)} \Leftrightarrow \text{s-an}: (\text{AN}; <\text{s-spec}\text{ident}_1;\text{ACC-1}> ) \]
\[ \text{T} \Leftrightarrow \text{null} \]

where:
\[ \text{ident}_1 = \text{s-ident}(\text{cond}) \]
\[ \text{attn}_1 = \text{ident}_1(\text{AN}) \]

(44) \[ \text{change-enab-1}(\text{cond}) = \]
\[ \text{is-ACC-1}\text{s-spec(\text{attn}_1)} \& \text{is}-<>\text{s-info}(\text{attn}_1) \rightarrow \]
\[ \text{s-an}: (\text{AN}; <\text{s-spec}\text{ident}_1;\text{ASYN}> ) \]
\[ \text{s-ra}: (\text{PA}; <\text{TN};\text{prep-attr-1}(\text{TN}(\text{PA}), \text{cond})> ) \]
\[ \text{is-ACC-1}\text{s-spec(\text{attn}_1)} \rightarrow \text{s-an}: (\text{AN}; <\text{s-spec}\text{ident}_1;\text{ASYN}> ) \]
\[ \text{T} \Leftrightarrow \text{null} \]

where:
\[ \text{ident}_1 = \text{s-ident}(\text{cond}) \]
\[ \text{attn}_1 = \text{ident}_1(\text{AN}) \]

10. ATTENTIONS AND CONDITIONS 15
10.3.2 ACCESS STATEMENT

The access statement makes available attention information for processing, either taking one attention of a specified list or an arbitrary one out of the list of enabled attentions in the task. If an attention is enabled in the task, but the attention stack is empty, then the else-unit is taken, or in its absence the task waits until information appears on the stack.

\[
\text{int-access-st}(t) =
\begin{align*}
\text{int-acc-1}(\text{cdl}, t); \\
\text{cdl} : \text{pass-} & \text{eval-} \text{attn-list} (s\text{-}\text{attn-list} \cdot s\text{-}\text{cond}(t), \text{DN}, \text{NV}) \\
\text{for: is-access-st}(t)
\end{align*}
\]

Ref.: eval-attn-list 10-7(15)

\[
\text{int-acc-1}(\text{cdl}, t) =
\begin{align*}
\text{is-<>}(\text{cdl}) & \rightarrow \text{int-acc-2}(t) \\
\text{test-attn}(\text{cdl}, \text{AN}, \text{EN}, \text{TW}) & \rightarrow \text{int-acc-3}(\text{cdl}, t)
\end{align*}
\]

for: is-access-st(t)

Ref.: is-access-st 10-7(15)

\[
\text{test-attn}(\text{cdl}, \text{an}, \text{en}, \text{tn}) =
\begin{align*}
\text{is-<>}(\text{cdl}) & \rightarrow T \\
(\text{tn}_1 = \text{tn} \ & \text{s-spec(attn}_1) = \text{ASYN}) \ & \text{s-spec(cond}_1) = \text{ASYN} & \rightarrow \text{error} \\
\text{tn}_1 & \rightarrow T \\
(\exists x)(1 \leq i \leq \text{length}(s\text{-assoc-list}(\text{en})) \ & x = \text{elem}(i, s\text{-assoc-list}(\text{en})) \ & \text{s-ident}(x) = \text{ident}_1) & \rightarrow \\
\text{test-attn}(\text{tail}(\text{cdl}), \text{an}, \text{en}, \text{tn}) \\
T & \rightarrow \text{error}
\end{align*}
\]

where:

\[
\begin{align*}
\text{ident}_1 & = \text{s-ident(head(\text{cdl}))} \\
\text{attn}_1 & = \text{ident}_1(\text{an}) \\
\text{tn}_1 & = \text{s-task(\text{attn}_1)} \\
\text{cond}_1 & = (\exists x)(31)(x = \text{elem}(i, s\text{-assoc-list}(\text{en})) \ & \text{1} \leq i \leq \text{length}(s\text{-assoc-list}(\text{en})) \ & \text{s-ident}(x) = \text{s-ident\text{-}head(\text{cdl}))}
\end{align*}
\]

Ref.: is-tn(3-4(6))
(49) \[ \text{\texttt{int-acc-3}(t) =} \]
\[ \text{\texttt{-is-eattn-cond(select-cond(cdl,AN,tn))} \rightarrow \text{call-attn-cond(select-cond(cdl,AN,tn))}} \]
\[ \text{\texttt{-is-eattn-cond(select-cond(cdl,AN,tn))} \rightarrow \text{stack-ci}(F,t)} \]
\[ T \rightarrow \text{wait-acc}(\text{cdl}) \]

where:
\[ \text{cdl} = \text{order-set}((\text{elem}(1,\text{s-enab-list}(\text{EN})) \mid 1 \leq i \leq \text{length}\text{s-enab-list}(\text{EN}) & \text{s-spec}\text{elem}(i,\text{s-enab-list}(\text{EN})) = \text{ACC}),\text{ACC}) \]

for: \text{is-access-st}(t)

Ref.: \text{order-set 11-41(111)}

(50) \[ \text{\texttt{int-acc-3}(cdl,t) =} \]
\[ \text{\texttt{-is-eattn-cond(select-cond(cdl,AN,tn))} \rightarrow \text{call-attn-cond(select-cond(cdl,AN,tn))}} \]
\[ \text{\texttt{-is-eattn-cond(select-cond(cdl,AN,tn))} \rightarrow \text{stack-ci}(F,t)} \]
\[ T \rightarrow \text{wait-acc}(\text{cdl}) \]

for: \text{is-access-st}(t)

Ref.: \text{is-eattn-cond 3-19(72)}
\text{stack-ci 6-39(109)}

Note: The instruction \text{stack-ci} interprets the else unit. It is necessary to use this instruction here to handle also goto statements within the on-unit correctly.

(50.1) \[ \text{select-cond(cdl,an,tn) =} \]
\[ \text{\texttt{-is<>(cdl)} \rightarrow F} \]
\[ \text{\texttt{s\text{-}task(attn} _1\text{)} = \text{tn} \text{ & s-spec(attn} _1\text{)} = \text{ACC} \text{ & -is<>s\text{-}info(attn} _1\text{)} \rightarrow \text{head(cdl)}} \]
\[ T \rightarrow \text{select-cond(tail(cdl),an,tn)} \]

where:
\[ \text{attn} _1 = (\text{s\text{-}ident\text{-head(cdl)\text{-}}})(\text{an}) \]

for: \text{is-tn}(tn)

Ref.: \text{is-tn 3-4(6)}
(52) \texttt{wait-acc}(cdl) = \\
\texttt{is-eattn-cond}(\texttt{select-cond}(cdl,AN,TN)) \rightarrow \texttt{call-attn-cond}(\texttt{select-cond}(cdl,AN,TN)) \\
T \rightarrow \\
\texttt{wait-acc}(cdl) ; \\
\texttt{set-wait-state-4}(cdl) \\
\texttt{Ref.: is-eattn-cond 3-19(72)}

(53) \texttt{set-wait-state-4}(cdl) = \\
\texttt{is-eattn-cond}(\texttt{select-cond}(cdl,AN,TN)) \rightarrow \texttt{call-attn-cond}(\texttt{select-cond}(cdl,AN,TN)) \\
T \rightarrow \texttt{s-te}:\mu(IE:\texttt{<s-wait:*>}) \\
\texttt{Ref.: is-eattn-cond 3-19(72)}

10.4 CONDITION ACTIVATION

If the interpretation of a program arrives in certain situations the language specifies that a condition is to be raised. Due to the heterogeneous sources of condition raising some of the condition raising actions will be described in the appropriate sections of the language definition and this chapter will give auxiliary instructions used for the condition activation.

10.4.1 INTERPRETATION OF THE CONDITION CALL

The instruction \texttt{call-cond-1} is the general form of a condition call, which is used when a cbif argument is required for updating the condition builtin function part of CS. In the other cases the instruction \texttt{call-cond} is used.

The instruction \texttt{call-cond-2} inspects the condition status for an appropriate condition action. If none is present or the on-unit is SYSTEM, \texttt{syst-cond-exec} is used to execute the system action, otherwise \texttt{int-cond} is used to execute the condition action.

(54) \texttt{call-cond}(cond) = \\
\texttt{call-cond-1}(\texttt{cond,\mu_0}(\langle s-oncode\ast s-cbif:code(\texttt{cond,\xi}\rangle))) \\
for: (\texttt{is-prefix-cond} \& \texttt{is-check-cond} \& \texttt{is-AREA} \& \texttt{is-ERROR} \& \texttt{is-FINISH})(\texttt{cond}) \\
\texttt{Ref.: code 10-27(80)}
31 October 1969

ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

rec-code(z-1,z-2) =

for:z-1 # z-2

Note: This function is implementation defined. Its value is an integer value indicating which of the both sizes is greater than the other, to be used for the ONCODE built-in function. The function is partially described by the following axiom.

11.5.1.2 Write transmission

test-and-write(et, skip) =

is-SKIP(skip) -> PASS: {} T -> write-transmission(et)

for:is-e-write-st(et) & (is-SKIP v is-0) (skip)

Ref.: is-e-[pred] 11-37(99)

write-transmission(et) =

-is-0(st_0) -> error T ->

PASS: cond-set([cond_1, tmt_0], f_0, s-ident(et))

where:

write_1 = write(mp_0, ds_0, rec_0)
rec_0 = μ₀(<s-key,s-ident(et)>,<s-vr:(s-pp->s-from(et)) (Ε)>)
cond_1 = (is-REC*s-inf(write_1) ->
rec-code(size-1*s-pp->s-from(et),size*s-vr*size-el(mp_0,ds_0,rec_0)),
T -> s-inf(write_1))

for:is-e-write-st(et)

Ref.: write 11-16(47)
  size-1 3-15(18)
  size 3-15(33)
  size-el 11-10(35)
  is-e-[pred] 11-37(99)

11. INPUT AND OUTPUT 57
In this section the allocation and initialization of the based variable specified in the locate statement is described. The allocation in main storage as well as the allocation in an area is closely related to the corresponding sections of chapter 7. Furthermore, the new pointer value is assigned to the explicit or implicit pointer variable, and the corresponding element of $PU$ is updated by the relevant information about the new buffer.

\begin{align*}
(152) \text{unite}(\text{set-1}, \text{set-2}) = \\
\quad \text{pass: set-1 u set-2} \\
\quad \text{for: is-set(set-1) \& is-set(set-2)}
\end{align*}

11.5.1.3 Buffer allocation

In this section the allocation and initialization of the based variable specified in the locate statement is described. The allocation in main storage as well as the allocation in an area is closely related to the corresponding sections of chapter 7. Furthermore, the new pointer value is assigned to the explicit or implicit pointer variable, and the corresponding element of $PU$ is updated by the relevant information about the new buffer.

\begin{align*}
(153) \text{test-and-allocate}(et, \text{skip}) = \\
\quad \text{is-SKIP(skip) \rightarrow \text{null}} \\
\quad T \rightarrow \text{allocate-buffer}(et) \\
\quad \text{for: is-e-locate-st(et) \& (is-SKIP v is-U)(skip)}
\end{align*}

Ref.: is-e-[pred] 11-37(99)

\begin{align*}
(153.1) \text{allocate-buffer}(et) = \\
\quad \text{is-based(attr)} \& \text{is-var-ref(ptr, A)} \& \text{is-PR-s-da-aggr-ref(ptr, A)} \rightarrow \\
\quad \text{initialize}(\text{gen, aggr 1}); \\
\quad \text{set-and-refer}(\text{ptr, gen, aggr 1}); \\
\quad \text{gen: allocate-based-buffer(eva, s-file(et), s-ident(et));} \\
\quad \text{eva: eval-alloc-aggr(aggr 1)} \\
\quad \text{is-based(attr) \& is-var-ref(ptr, A) \& is-var-ref(area, A)} \rightarrow \\
\quad \text{initialize}(\text{gen, aggr}); \\
\quad \text{set-and-refer}(\text{ptr, gen, aggr}); \\
\quad \text{gen: allocate-area-buffer(gen-1, eva, area, s-file(et), s-ident(et));} \\
\quad \text{eva: eval-alloc-aggr(aggr 1),} \\
\quad \text{gen-1: eval-ref-gen(area 1)} \\
\quad T \rightarrow \text{error}
\end{align*}

where:
\begin{align*}
\text{n} &= \text{s-n(et)} \\
\text{attr} &= \text{n}_{1}(A) \\
\text{aggr} &= \text{s-aggr(attr)} \\
\text{ptr} &= (\text{is-\text{os-ptr(et)} \rightarrow \text{s-ptr(et),}} \\
\quad T \rightarrow \text{s-\text{ptr(attr)}} \\
\text{area} &= \text{s-area \& s-da-aggr-ref(ptr, A)}
\end{align*}

\begin{align*}
\text{for: is-e-locate-st(et)}
\end{align*}

Ref.: is-var-ref 8-9(20.1) \\
\text{aggr-ref 8-25(70.1)} \\
\text{initialize 7-15(49)}

cont'd