PL/IDE GRAMMAR

P. S. Newman and I. S. Zukerman

Printed 07/20/83

```
1   <RELOP> ::= EQ
2             | NEQ
3             | GT
4             | LT
5             | GE
6             | LE
7   <OROP> ::= OR
8   <ANDOP> ::= AND
9   <NOTOP> ::= NOT
10  <LOGICAL VALUE> ::= TRUE
11                    | FALSE
12  <SET RELOP> ::= EQS
13                | NEQS
14                | IN
15                | NOTIN
16                | CTNS
17  <SETOP> ::= UNION
18            | INTER
19            | MINUS
20  <NULL> ::= NULL
21  <ADDOP> ::= +
22            | -
23  <MULTOP> ::= *
24             | /
25  <EXPOP> ::= **
26  <CONCATENATION OP> ::= ||
27  <ASSIGNMENT OP> ::= =
28                    | +=
29                    | -=
30  <QUESTION MARK> ::= ?
31  <COLON> ::= :
32  <LEFT TUPLE BR> ::= <
33  <RIGHT TUPLE BR> ::= >
34  <LEFT QUERY BR> ::= {
35  <RIGHT QUERY BR> ::= }
36  <LEFT SET BR> ::= {*
37  <RIGHT SET BR> ::= *}
38  <LEFT AGG BR> ::= [
39  <RIGHT AGG BR> ::= ]
40  <LEFT TEMPLATE BR> ::= [*
41  <RIGHT TEMPLATE BR> ::= *]
42  <SUBLIST SEP> ::= //
43  <DONT CARE> ::= *
44  <ARGUMENT PREFIX> ::= |
45  <DOT> ::= .
46  <PCT> ::= %
47  <NONEXISTENT> ::= ¢
48  <RELPREFIX> ::= ::
```

```
223   <SET> ::= <LEFT SET BR> <SET LIST> <RIGHT SET BR>
224           | <LEFT SET BR> <RIGHT SET BR>
225   <SET LIST> ::= <EXPRESSION>
226                | <SET LIST> , <EXPRESSION>
227   <TUPLE> ::= <LEFT TUPLE BR> <TUPLE LIST> <RIGHT TUPLE BR>
228   <TUPLE LIST> ::= <TUPLE ELEMENT>
229                  | <TUPLE LIST> , <TUPLE ELEMENT>
230   <TUPLE ELEMENT> ::= <EXPRESSION>
231                     | <DONT CARE>
                        | <QUESTION MARK>
```

A <SET> has the following format: {* exp1, exp2,...,expn *} where n>=0 . The language does not have sets of sets, therefore a semantic check has to be performed in order to verify that expi 1<=i<=n evaluates to a scalar or a tuple.

A <TUPLE> has the following format: < element1,...,elementn > where n>=1.

A <TUPLE ELEMENT> may be either an expression (which evaluates to a tuple or a scalar), a don't care symbol ('*') or a question mark symbol ('?'). The don't care symbol may be used in set relational operations. In this case, it means that the value of the attribute in the corresponding column is not important to the result of the comparison.
The question mark symbol can be used as a tuple element only at particular points within the BASIC AGGREGATE CONSTRUCTOR (see explanation on aggregates below).

The following examples will illustrate the usage of the above described constructs:

```
1.  {* A+B,13,2**X *}
2.  {* <D,F>,<X,Z> *}
3.  < "ABC",<"DEF",F(X)> >
```

The first example is straightforward. The second example represents a set, whose members are tuples.

The third example represents a tuple, whose first member is a string and whose second member is itself a tuple.

```
163   <EXPRESSION> ::= <BOOLEAN TERM>
164                  | <EXPRESSION> <OROP> <BOOLEAN TERM>
165   <BOOLEAN TERM> ::= <BOOLEAN FACTOR>
166                    | <BOOLEAN TERM> <ANDOP> <BOOLEAN FACTOR>
167   <BOOLEAN FACTOR> ::= <SET RELATION>
168                      | <NOTOP> <SET RELATION>
169   <SET RELATION> ::= <RELATION>
170                  | <PREDICATE ABBREVIATION>
171                  | <SET RELATION> <SET RELOP> <RELATION>
172                  | <SET RELATION> <SET RELOP> <PREDICATE ABBREVIATION>
173   <RELATION> ::= <SET EXP>
174                | <SET EXP> <RELOP> <SET EXP>
175   <SET EXP> ::= <ARITHMETIC EXP>
176               | <SET EXP> <SETOP> <ARITHMETIC EXP>
179   <ARITHMETIC EXP> ::= <TERM>
180                      | <ARITHMETIC EXP> <ADDOP> <TERM>
181   <TERM> ::= <FACTOR>
182            | <TERM> <MULTOP> <FACTOR>
183   <FACTOR> ::= <SIGNED PRIMARY>
184              | <FACTOR> <EXPOP> <SIGNED PRIMARY>
185   <SIGNED PRIMARY> ::= <CHAR EXP>
186                      | <ADDOP> <CHAR EXP>
187   <CHAR EXP> ::= <PRIMARY>
188                | <CHAR EXP> <CONCATENATION OP> <PRIMARY>
189   <PRIMARY> ::= <UNSIGNED NUMBER>
190               | ( <EXPRESSION> )
191               | <REFERENCE>
192               | <SELECTION VAR>
193               | <EXIST QUANT>
194               | <LOGICAL VALUE>
195               | <CHARSTRNG>
196               | <SET>
197               | <TUPLE>
198               | <QUERY SET>
199               | <NULL>
200   <PREDICATE ABBREVIATION> ::= <IDENTIFIER> <COLON> <TUPLE>
201                              | <IDENTIFIER> <DOT> <IDENTIFIER> <COLON> <TUPLE>
202   <SELECTION VAR> ::= <QUESTION MARK> <IDENTIFIER>
203   <EXIST QUANT> ::= <QUESTION MARK> <QUESTION MARK> <IDENTIFIER>
204   <UNSIGNED NUMBER> ::= <INTEGER>
                          | <REAL>
```

Before launching into the explanation of the simple expressions of PL/IDE it should be noted that the above given BNF does not express type considerations. It only represents the basic skeleton of legitimate PL/IDE expressions.

PL/IDE contains arithmetic and boolean expressions present in most languages. The salient features in PL/IDE expressions are:

- set expressions - contain operations of union, intersection and substraction of sets. These operations have equal precedence and are performed from left to right.

- Set relational expressions - contain comparison operations between sets. The following comparison operations are featured in this language:

  - A CTNS B (containment),
  - A IN B (membership),
  - A NOTIN B,
  - A EQS B (set equality)
  - A NEQS B (set inequality)

  The main difference in the usage of set relational operators and "simple" relational operators (e.g: GT, LT, etc...) is that the former can appear in sequence in a meaningful expression, whereas the latter can appear only once. This, because relational operators expect arithmetic or string operands, whereas set-relational operators accept any sets (or scalars) as operands (as long as they are compatible in type). The following example will illustrate this point:

  ```
  A CTNS B IN C
  A GT B LT C
  ```

  In the first expression the user wants to check whether A contains B and then check whether the result (TRUE or FALSE) is in C (where C is a boolean set). The second expression is meaningless.

- Predicate abbreviations - this construct represents a limited and abbreviated version of the membership set-relational operation. As with relational operations, the result of a predicate abbreviation is a boolean. Notice that this construct requires a tuple after the colon. However this does not constrain predicate abbreviations to be used with tuples only, since tuple brackets may enclose an arbitrary expression as long as it evaluates to a scalar or tuple (see the above given explanation on tuples). The following example illustrates the usage of this construct:

  ```
  X AND A:<Y,5>
  where  A:<Y,5>   is equivalent to   <Y,5> IN A
  ```

Due to the introduction of set expressions, the rules of precedence between operators in PL/IDE differ from the rules of precedence in other languages. The following table represents the rules of precedence in PL/IDE:

precedence	type	operator
high	string	concatenation ||
	unary additive	unary + -
	exponential	**
	multiplicative	* /
	additive	binary + -
	set	UNION INTER MINUS
	relational	EQ NEQ GT GE LT LE
	set relational	EQS NEQS CTNS IN NOTIN
	boolean primary	NOT
	boolean	AND
low	boolean	OR

These rules of precedence apply only for type-compatible operations (i.e: operations between arithmetic and string values are not allowed). The arithmetic, string and boolean operators accept as operands only scalars (or sets **defined** as unary sets) of numeric, character and boolean type respectively. The set and set-relational operators accept as operands sets, tuples or scalars of compatible type. The relational operators accept either numeric or character string operators.

The following basic constructs (primaries) do not appear in other languages and bear some explanation:

- selection variables - these are the variables used in iteration structures and query sets (see explanation below). They can not appear on the left hand side of an assignment, and values can only be implicitly assigned to them during iteration or query set construction.

- existential quantifiers - used in "traditional" database queries,

- sets, tuples (explained above) and query sets (see explanation further on).

The following examples illustrate the usage of expressions in PL/IDE:

1. A GT C||"ABCD"
2. X+5 UNION B
3. B UNION C INTER D
4. X GT Y EQS M LT N
5. F LE G NEQS C NOTIN D

The first example shows a comparison operation between two character strings, namely A and the string resulting from the concatenation of C with "ABCD". The concatenation takes precedence over the relational operation. Like in other languages, string comparison is performed in lexicographic order.

The second example features an arithmetic operation (between scalars) followed by a set operation. In this case the result of X+5 is appended to the set B (if not already there).

The third example shows a sequence of set operations. As mentioned above, the operations are performed from left to right.

Examples 4 and 5 are of special interest: in example 4 the scalar relational operations (GT and LT) take precedence over the set relational operation (EQS). Thus, the two scalar operations are performed first and the result is compared by means of the EQS operation. In example 5 the scalar relational operation is performed first and the set relational operations (NEQS and NOTIN) are performed from left to right. Thus, although examples 4 and 5 present expressions similar in syntax, their semantics are different.

```
205   <REFERENCE> ::= <REFID> <ACTUAL ARGUMENT PART>
206                 | <REF PREFIX> <REFID> <ACTUAL ARGUMENT PART>
207                 | <REFID>
208   <REF PREFIX> ::= T'
209                  | I'
210   <ACTUAL ARGUMENT PART> ::= ( )
211                            | ( <ACTUAL ARGUMENT LIST> )
212   <ACTUAL ARGUMENT LIST> ::= <ACTUAL ARGUMENT>
213                        | <ACTUAL ARGUMENT LIST> , <ACTUAL ARGUMENT>
214   <ACTUAL ARGUMENT> ::= <QUESTION MARK>
215                       | <DONT CARE>
216                       | <ARGUMENT PREFIX> <EXPRESSION>
217                       | <ARG EXPRESSION>
235   <ARG EXPRESSION> ::= <EXPRESSION>
236                      | <AGGCONSTR>
                         | <OBJECT ID>
237                      | <EXTRACTOR> <LIM PART>
173   <LIM PART> ::=
174                | ( <EXPRESSION> )
220   <REFID> ::= <IDENTIFIER>
221             | <IDENTIFIER> <DOT> <IDENTIFIER>
222             | <IDENTIFIER> <PCT> <IDENTIFIER>
223   <OBJECT ID> ::= <IDENTIFIER> <DOT> <DOT>
```

A <REFERENCE> is a construct that represents either a reference to a literal set or a functional request or invocation.

A legitimate <REFID> (reference identifier) may be one of the following:

- <identifier>
- <identifier>.<identifier> (see explanation on REQUEST)
- <identifier>%<identifier> (see explanation on INVOKE)

A reference identifier may be followed by an argument part, which may contain an arbitrary number of arguments.

Conceptually there are three kinds of references:

- Local set references - which refer to sets local to the current module,
- Standard function references - which are references as we know them from other programming languages, and
- Literal function references - which may refer to one of the following:
  - Local sets followed by argument lists, or

- Functions which accept no arguments, and return sets, followed by argument lists. A literal function reference implies that the returned set is applied as a function to the argument list.

A standard function reference may include as arguments: expressions, <AGGREGATE CONSTRUCTOR>s, <AGGREGATE EXTRACTOR>s optionally followed by a <LIM PART> (see explanation below) or <OBJECT IDENTIFIER>s. The latter is a dereferenced pointer to an object (which is a copy of the object itself, as opposed to a pointer to the object). Arguments in PL/IDE are passed by value. An argument may be prefixed by the following symbol: "|" (termed "argument prefix"). This means that if the formal parameter is a scalar and the actual argument is a set, the function reference is repeated for each member of the set. In this case, the result is the union of the returned sets. The following tables will better illustrate this concept:

1. F(a)

formal / actual	scalar	set
scalar	o.k	set of one
set	error	o.k

2. F(|a)

formal / actual	scalar	set
scalar	o.k	set of one
set	repeat	o.k

3. F(|a1,a2,...,|ai,...,an) - option No. 2 is activated for the prefixed arguments and option No. 1 is activated for the rest.

In this manner the user can perform n1*...*ni*... function activations in a single call (where ni is the number of elements in argument i 1<=i<=n , and argument i is prefixed).

A standard function reference can be modified in the following way:
T'<reference identifier> - returns a set of tuples:

<i,"expected result of the function">

where i is a different integer for each tuple. Because of the added integer, the modified function will "repeat" all the identical elements in the "expected result of the function" part. This modifier is useful in calculating statistical functions for which duplicates are necesary (e.g: AVG, SUM, etc).

A literal function reference may be envisioned as a somewhat restricted selection-projection operation performed on a database relation. In this context, each argument position corresponds to an attribute in the relation.

In order to select all the tuples for which attribute i has certain values, these values have to be entered in the position corresponding to attribute i. Any valid set expression may be entered as such an argument, however aggregate constructors and extractors are not allowed in this context (since the language does not support sets of aggregates). The arguments can be optionally prefixed with the argument prefix symbol, however for literal function references the function will be repeated even if the arguments are not prefixed.

A projection is performed by entering a question mark ("?") for each attribute to be projected and a "don't care" symbol ("*") for each attribute to be eliminated from the final result. The user may terminate the function designator at any point by entering a right parenthesis. This causes the remaining attributes to be projected.

A literal function reference can be modified by the following prefixes:

- T'<reference identifier> - same as for standard function references, and

- I'<reference identifier> - returns the inverse of the normal referent. This modifier may be applied only to functions returning sets of binary tuples. The result is actually a reordering of the arguments.

The following examples will illustrate the usage of this construct:

```
1.  VAR1
2.  T'COST(X,Y,Z)
3.  PART_SUPPLIER("12345",?,*,?,"ELM ST. 10024")
4.  PART_SUPPLIER( {* "12345","56789","34567" *}, * )
5.  COST( |{* "12345","56789","34567" *},Y,Z )
```

The first example is straightforward. The second example features a functional invocation of COST, which expects three scalar parameters and normally returns a single salary. The illustrated reference returns a tuple, due to the presence of the prefix "T".

The third and fourth example feature a literal function reference to PART_SUPPLIER. This function returns a relation whose first attribute is part-number and whose last attribute is street-address (the other attributes are irrelevant). In example 3 the user requests the second and fourth attribute of those tuples whose part-number is "12345" and whose street-address is "ELM ST. 10024". Notice that the third attribute is to be omitted. In example 4 the user requests the third, fourth and last attribute of the tuples whose part-number is "12345" or "56789" or "34567". Recall that for literal function references the function activation is repeated for each member of the set.

The fifth example presents a reference to the COST object from example 2. However in this case, the first argument is a set prefixed by a "|". This means that COST is to be activated once for each member of the set. (If the argument prefix were omitted the system would respond with an error message, since COST expects a scalar as its first argument.)

```
232   <QUERY SET> ::= <LEFT QUERY BR> <QUERY> <RIGHT QUERY BR>
233   <QUERY> ::= <FROM BGN>
234               | <FROM BGN> WHERE <EXPRESSION>
235               | <QUERY BGN> WHERE <EXPRESSION>
236   <FROM BGN> ::= <QUERY BGN> FROM <REFERENCE>
237   <QUERY BGN> ::= <QUERY QUAL> <QUERY BASE>
238                   | <QUERY BASE>
239   <QUERY QUAL> ::= ALL
240                    | EACH
241                    | ANY
242                    | THE
243   <QUERY BASE> ::= <REFERENCE>
244                    | <TUPLE>
245                    | <SELECTION VAR>
```

A <QUERY SET> is a special kind of expression, which when evaluated returns a set. A query set can be used wherever an explicit set could be used. The general format of a query set is:

```
{ element1   FROM  <function designator>   WHERE  <expression> }
```

Either the FROM clause or the WHERE clause may be omitted, however at least one of them has to appear in a query set. (Note that the FROM clause is an abbreviation of the WHERE clause, since " x FROM A " is equivalent to " x WHERE x IN A " .) The expression in the WHERE clause has to be boolean.

element1 has to be a single item, e.g: a <SELECTION VARIABLE>, a <REFERENCE> or a <TUPLE>. This is to alert the user to the fact that element1 has to be a legitimate member of a set. Notice that this syntax still enables the user to enter arbitrary expressions in place of element1, provided they are enclosed by tuple brackets. The user also has to ensure that element1 contains at least one selection variable.

element1 may be prefixed by the following qualifiers: ALL or EACH (this is the default and implies the return of all qualifying results), ANY (implies one arbitrarily chosen qualifying result) and THE (implies that the result set contains only one member).

The following examples will illustrate the use of this construct:

```
1.  { <?X,?Y> FROM DB1.A WHERE G(?X) GT 5 AND ?Y LT 8 }
2.  { EACH <?X+10> WHERE ?X IN { ?Y FROM DB2.C } }
3.  <?X,?Y,?Z> WHERE <?X,?Y,??W> IN SETA AND FUNC1:<?Z,??W>
```

In the first example, the user requests all tuples from set A in database DB1 which fulfill the condition in the WHERE clause. Of special interest is the second example, which contains nested query sets.

The last example can be best rephrased as follows: all tuples of the form <?X,?Y,?Z> where there exists a value (??W) such that <?X,?Y,??W> belongs to SETA and <?Z,??W> is a tuple in the literal function FUNC1. Note the use of selection variables to **extract** the required tuples.

Aggregate expressions enable the user to perform operations on **collections** of sets. With this construct, several sets may be grouped under one name, and one set may be part of different aggregates. There are three types of aggregate expressions:

- aggregate constructors - explained in the first subsection,
- aggregate extractors - explained in the second subsection, and
- references returning aggregates - which are functional references as seen above, whose result is one aggregate (implicitly returned), or references to declared paths (which are implicit extractors).

Aggregate extractors and references returning aggregates may participate in assignment statements (both on the right and left hand side). Whereas aggregate constructors can only appear in the right hand side of assignment statements. All three forms may be passed as arguments and may also appear as operands to the UNION, INTER and MINUS operators. Before elaborating on the usages of these expressions, a description is in order.

AGGREGATE CONSTRUCTORS

```
136  <AGGCONSTR> ::= <LEFT AGG BR> <BASICAGG> <RIGHT AGG BR>
137                | <LEFT TEMPLATE BR> <TEMPLATEDAGG> <RIGHT TEMPLATE BR>
138  <BASICAGG> ::= <IDENTIFIER> <ELEMENTLIST>
139                | <BASICAGG> <IDENTIFIER> <ELEMENTLIST>
140  <ELEMENTLIST> ::= <AGG EXPRESSION> <BASICAGG CONTINUATION>
141          | <ELEMENTLIST> , <AGG EXPRESSION> <BASICAGG CONTINUATION>
142  <BASICAGG CONTINUATION> ::=
143                        | <LEFT AGG BR> <BASICAGGCONSTR> <RIGHT AGG BR>
144  <BASICAGGCONSTR> ::= <RELPREFIX> <IDENTIFIER> <ELEMENTLIST>
145            | <IDENTIFIER> <ELEMENTLIST>
146            | <BASICAGGCONSTR> <RELPREFIX> <IDENTIFIER> <ELEMENTLIST>
147            | <BASICAGGCONSTR> <IDENTIFIER> <ELEMENTLIST>
148  <AGG EXPRESSION> ::= <EXPRESSION>
149                      | <QUERY>
150  <TEMPLATEDAGG> ::= <RELIDLIST> <SUBLIST SEP> <TEMPLATEDLIST>
151  <RELIDLIST> ::= <IDENTIFIER> <RELIDHEADER>
152                | <RELIDLIST> <IDENTIFIER> <RELIDHEADER>
153  <RELIDHEADER> ::=
154             | <LEFT TEMPLATE BR> <RELIDSUBLIST> <RIGHT TEMPLATE BR>
155  <RELIDSUBLIST> ::= <RELPREFIX> <IDENTIFIER> <RELIDHEADER>
156              | <IDENTIFIER> <RELIDHEADER>
157              | <RELIDSUBLIST> <RELPREFIX> <IDENTIFIER> <RELIDHEADER>
158              | <RELIDSUBLIST> <IDENTIFIER> <RELIDHEADER>
     <TEMPLATEDLIST> ::= <TEMPLATEDRELIST>
160                    | <TEMPLATEDLIST> <TEMPLATEDRELIST>
161  <TEMPLATEDRELIST> ::= <TEMPLATEDELEMENTLIST>
162                      | <NONEXISTENT>
163  <TEMPLATEDELEMENTLIST> ::= <TEMPLATED PRIMARY> <TEMPLATEDAGGCONT>
164   | <TEMPLATEDELEMENTLIST> , <TEMPLATED PRIMARY> <TEMPLATEDAGGCONT>
165  <TEMPLATEDAGGCONT> ::=
166       | <LEFT TEMPLATE BR> <TEMPLATEDAGGCONSTR> <RIGHT TEMPLATE BR>
167  <TEMPLATEDAGGCONSTR> ::= <TEMPLATEDRELIST>
168                 | <RELPREFIX> <TEMPLATEDRELIST>
169                 | <TEMPLATEDAGGCONSTR> <TEMPLATEDRELIST>
170                 | <TEMPLATEDAGGCONSTR> <RELPREFIX> <TEMPLATEDRELIST>
171  <TEMPLATED PRIMARY> ::= <TEMPLATED NUMBER>
172                        | <CHARSTRNG>
173                        | <TEMPLATED TUPLE>
174  <TEMPLATED NUMBER> ::= <UNSIGNED NUMBER>
175                       | <ADDOP> <UNSIGNED NUMBER>
176  <TEMPLATED TUPLE> ::= <LEFT TUPLE BR> <TEMPLATED TUPLELIST>
                                                  <RIGHT TUPLE BR>
177  <TEMPLATED TUPLELIST> ::= <TEMPLATED PRIMARY>
178                       | <TEMPLATED TUPLELIST> , <TEMPLATED PRIMARY>
```

The basic aggregate constructor <BASICAGG> establishes a temporary unnamed aggregate by defining its sets and assigning values to them. The use of this construct is best demonstrated by an example:

```
[ AGENT "Ann E."
        [ HAS_TERR "N.Y" , "N.J"
          HAS_COMM .10 [ COMM_TYPE "A" ::TDATE "9/82" ]
        ],
       "Bob M." [ HAS_TERR "Pa." ]
]
```

This aggregate constructor builds the following sets:

```
 AGENT          HAS_TERR              HAS_COMM
"Ann E."    <"Ann E.","N.Y">      <"Ann E.",.10>
"Bob M."    <"Ann E.","N.J">
            <"Bob M.","Pa.">          COMM_TYPE
                                      <.10,"A">
                  TDATE
        <<"Ann E.",.10>,"9/82">
```

The following facts should be noted in this example:

- A data element can be either an arbitrary expression or a query (which is a query set without the braces). In general, an expression in an aggregate constructor, has to evaluate to a scalar. However, a set expression can be introduced at any "leaf" in the aggregate constructor. In this case, it precludes further modifications of the aggregate, and it is equivalent to an assignment statement. The following expression illustrates this point:

  ```
  [ A  X UNION Y  ]  is equivalent to  A = X UNION Y
  ```

  When a data element is a query, the user can create an aggregate by means of a single (iterative) expression. For example, the following expression

  ```
  [ A  EACH ?T FROM DB1.A
             [  R1 DB1.R1(?T) R2 DB1.R2(?T)  ]
  ]
  ```

  puts in set A all tuples from set DB1.A. It creates set R1 from the join of set A and set DB1.R1 and finally it creates set R2 from the join of set A and set DB1.R2.

- Elements of sets are separated by commas. If an element is "modified" by an aggregate constructor, a comma still has to be inserted before the next element (e.g: between "Ann E." and "Bob M.").

- The last element of a set is followed either by a right aggregate brace (']'), or by the name of the next set.

- If a set name S is embedded in a hierarchy (like COMM_TYPE), it must be the name of a tuple-set. This means that the rightmost value assigned to S1 (immediately above set S in the hierarchy), is to participate in the tuples being assigned to set S.

In order to create a set S, whose tuples contain the last **tuple** assigned to S1 (formed by values cascading from the top of the hierarchy and the rightmost value), the set name S should be prefixed by a double colon (like in ::TDATE).

The basic aggregate constructor may be also used to create "n-tuples". The following example will best illustrate this usage:

```
[ WAREHOUSE "N.J"  [ PWQ  <"P1",?,5> , <"P2",?,7>  ]  ]
```

This aggregate constructor builds the following sets:

```
WAREHOUSE          PWQ  (Part-Warehouse-Quantity)
  "N.J"        <"P1","N.J",5>
               <"P2","N.J",7>
```

As stated above, tuples with question marks are only allowed in basic aggregate constructors. Notice also, that no double colons should be used when building "n-tuples", since they are meaningless in this context.

The templated aggregate constructor <TEMPLATEDAGG> is more succint than the basic aggregate constructor. It features a header which contains a **template** of set names. This header is followed by the values assigned to those names. The correspondence between set names and values is positional. Thus the user can perform bulk loading by means of this construct. The above given example will be now repeated in templated form:

```
[* AGENT   [  HAS_TERR  HAS_COMM [ COMM_TYPE ::TDATE  ] ]
// "Ann E." [ "N.Y","N.J"  .10    [    "A"    ::"9/82" ] ],
   "Bob M." [  "Pa." ]
*]
```

The templated constructor differs from the basic constructor in a few details:

- Only data primitives are allowed in this construct, i.e: numbers, character strings and tuples (which contain numbers and/or character strings only). This is because in bulk loading the data usually is of this form.
- Since the data is entered in bulk, a device to signal absence of data is necessary. This function is performed by the symbol "¢".
- In order to enforce consistency in the user, a double colon is required both in the header and in the data.

AGGREGATE EXTRACTORS

```
179   <EXTRACTOR> ::= <DOT> <LEFT AGG BR> <RELIDSUBLIST> <RIGHT AGG BR>
180     | <IDENTIFIER> <DOT> <LEFT AGG BR> <RELIDSUBLIST> <RIGHT AGG BR>
```

Aggregate <EXTRACTOR>s are complex object references that return aggregates.
The basic structure of an aggregate extractor is the same as the structure of the HEADER of the templated aggregate constructor (see <TEMPLATEDAGG>. above). Notice however, that the extractor is preceded by a dot, and optionally by an identifier (which is the name of the object where the sets mentioned in the <RELIDSUBLIST> are declared). When the identifier is absent, it means that these sets are local to the current module.
When the <LIM PART> is present, the leftmost set in the extractor is intersected with the expression in the <LIM PART>. This form is valid only when the extractor is passed as an argument or used on the right hand side of an assignment statement.

The meaning of this construct is best conveyed by an example:

```
DB1.[ A [ B C [ D ] ] ]
```

Set A is first extracted from object DB1. Then the sets

```
A natural_join B     and     A natural_join C
```

named B and C respectively are built. Notice that the type of the first "column" in sets B and C has to be compatible with the type of set A.
Finally the set C natural_join D (named D) is constructed.

The user should notice that aggregate extractors are an abbreviated version of the above seen basic aggregate constructors for some special cases. The following example will illustrate this point:

```
DB1.[  A  [  B C  ]  ]  ( exp )
is equivalent to
[ A  EACH ?T IN (DB1.A INTER exp )
        [ B  DB1.B(?T)  C DB1.C(?T)  ]
]
```

Although it was stated above that extractors can be used on the left hand side of assigment statements, it should be noted that only flat explicit extractors or declared flat paths can be used in such a context.

USAGE OF AGGREGATE EXPRESSIONS

Now that the user is more familiar with these expressions, some remarks regarding their usage are necessary.
As stated above, aggregate expressions may be used in the following instances:

- As operands of set operators (UNION, INTER and MINUS) - in this case, sets of the same name are located in the participating aggregates, and the specified operation is performed between matching sets. The remaining sets are part of the resulting aggregate for a UNION opera-

tion or are eliminated from the result for an INTER operation. For a MINUS operation only the "unmatched" sets of the left operand appear in the result.
For example, if we perform these set operations on the following aggregates:

```
AGG1 = DB1.[ A B C ]
AGG2 = DB2.[ D C A ]
```

where DB1 and DB2 are two different objects.

The resulting aggregate will contain the following sets:

operation	UNION	INTER	MINUS
sets in resulting aggregate	DB1.A UNION DB2.A DB1.C UNION DB2.C DB1.B DB2.D	DB1.A INTER DB2.A DB1.C INTER DB2.C	DB1.A MINUS DB2.A DB1.C MINUS DB2.C DB1.B

- In assignment statements - explained later on (see CONTROL STATEMENTS), and
- As arguments to a reference - in this case the name of the aggregate may be passed as the argument, however the target reference should be able to operate directly on the sets in the aggregate (with the set names **locally** defined).

Notice that in all these uses of aggregates, the names of the sets inside the aggregates are crucial to the outcome of the specified operation. Thus, in order to perfom an operation between sets with different names (or **not** to perform an operation between equally named sets), the user should be able to "change" the name of the set for the duration of the operation. This can be done by a renaming built-in function.

The currently featured renaming function has the following structure:

```
RENAME(aggregate_identifier,renaming set)
```

where the renaming set is either an expression that returns a set or an explicit set of tuples of the following form:

```
{* <oldname1,newname1>,<oldname2,newname2>,... *}
```

where oldname is the current name of the set and newname is the name to be used during the operation. If the set oldname does not exist in the aggre-

gate, the corresponding tuple is ignored. Conversely, if newname is a name already in use by the current module, an error message is issued.

The renaming function may be also used to the right of functional references returning aggregates.

We will now perform the above demonstrated INTER operation between AGG1 and AGG2 with the help of the renaming function:

```
AGG1 INTER RENAME(AGG2,{* <A,X>,<D,B> *})
```

In this example, set A in AGG2 has been renamed to X (thus precluding its participation in the intersection operation) and set D in AGG2 has been renamed to B (achieving the opposite result). The result of the intersection will be an aggregate containing the following sets:

```
DB1.B INTER DB2.D
DB1.C INTER DB2.C
```

```
 81  <RETURN STMT> ::= RETURN <RETURN PART> <REENTRY PART>
 82                  | RETURN <REENTRY PART>
 83                  | RETURN <IDENTIFIER> <RETURN PART> <REENTRY PART>
 84                  | RETURN <IDENTIFIER> <REENTRY PART>
 85  <RETURN PART> ::= ( )
 86                  | ( <RETURN LIST> )
 87  <RETURN LIST> ::= <ARG EXPRESSION>
 88                  | <RETURN LIST> , <ARG EXPRESSION>
 89  <REENTRY PART> ::=
 90                   | REENTRY <IDENTIFIER>
 91  <SEND STMT> ::= SENDF <REFERENCE> <VIA PART>
 92                | SENDL <REFERENCE> <VIA PART>
 93  <VIA PART> ::=
 94               | VIA <EXPRESSION>
 95  <ASYNC RCV STMT> ::=  RCV <QPROC> ; <RCV ON PART> <END STMT>
 96            | RCV <QPROC> <SELECTION VAR> ; <RCV ON PART> <END STMT>
 97  <QPROC> ::=
 98            | FIRST
100            | SEQ
101  <RCV ON PART> ::=
102                 | <RCV ON STMT LIST> ;
103  <RCV ON STMT LIST> ::= <RCV ON STMT>
104                      | <RCV ON STMT LIST> ; <RCV ON STMT>
105  <RCV ON STMT> ::= ONL <IDENTIFIER> <ON CONTINUATION>
106            | ONF <IDENTIFIER> <DOT> <IDENTIFIER> <ON CONTINUATION>
107  <ON CONTINUATION> ::= ( <TARGET PART> )
108                     | ( <TARGET PART> ) WHERE <EXPRESSION>
109                     | ( <TARGET PART> ) THEN <UNLABELLED BASIC STMT>
110                     | ( <TARGET PART> ) WHERE <EXPRESSION>
                                              THEN <UNLABELLED BASIC STMT>
111  <TARGET PART> ::=
112                 | <TARGET LIST>
113  <SYNC RCV STMT> ::= SRCV ; <SRCV ON> <END STMT>
123  <ERCV STMT> ::= ERCV ( <TARGET PART> )
                   | ERCV <IDENTIFIER> ( <TARGET PART> )
114  <SRCV ON> ::=
115              | <ON STMT LIST> ;
116  <ON STMT LIST> ::= <ON STMT>
117                   | <ON STMT LIST> ; <ON STMT>
118  <REQUEST STMT> ::= <TRANS> REQ <REQ ID> ; <SRCV ON> <END STMT>
     <REQ ID> ::= <IDENTIFIER> <DOT>
                | <IDENTIFIER> <DOT> <IDENTIFIER>
119  <ON STMT> ::= ON <IDENTIFIER> ( <TARGET PART> )
120    | ON <IDENTIFIER> ( <TARGET PART> ) THEN <UNLABELLED BASIC STMT>
121  <INVOKE STMT> ::= INVOKE <REFERENCE> ; <SRCV ON> <END STMT>
122  <SHORT ARGS> ::= ( <SUBLIST SEP> <TARGET PART> )
123            | ( <ACTUAL ARGUMENT LIST> <SUBLIST SEP> <TARGET PART> )
```

The <RETURN STMT> transfers control from the current module to the calling module. It returns a list of set and/or aggregate expressions, whose types must match those declared in association with the return message class (the <IDENTIFIER> in RETURN <IDENTIFIER>). If the entire object has only one return message class, it need not be explicitly specified in the RETURN statement. When the <REENTRY PART> is added to the return statement, it means that upon the next activation of the current module, execution will start at the statement indicated in the reentry part.

The <SEND STMT> is used for asynchronous communication. The SENDF statement sends a list of arguments whose type is defined by a "message class" of the recipient, whereas the SENDL statement sends arguments whose type is defined by a message class of the sender. The VIA part may be omitted, however when it is appended to the statement, it is used to identify a message as part of a specific conversation.

There are two kinds of receive statements: synchronous and asynchronous. Both of them have the basic structure of a case statement: when a message is received, different ON statements are executed, depending on the "message class" accompanying the message. The execution of an ON statement implies passing the arguments of the message to the formal parameters of the target statement and then executing its "THEN" part.

An asynchronous receive statement <ASYNC RCV STMT> has a header and an ON-part.

The header may contain a selection variable, which receives the identifier of the message. (Notice that one conversation may contain many messages.) The header may also contain a directive regarding the order in which messages in the queue are to be considered for acceptance:

- FIRST - choose the message which matches the earliest ON statement,
- SEQ - (default option) choose the message according to FIFO (First In First Out) policy.

The ON part may contain local or foreign ON statements, ONL and ONF respectively. In the former, only the name of the message class is necessary. In the latter, both the name of the message class and the name of the module in which the message class is defined, have to be stated.

An ON statement may be accompanied by a guard expression (in the form of a WHERE clause), which specifies conditions necessary for the execution of the statement. A message in the queue satisfies a case if it is of the expected message class and the guard expression is fulfilled. If the first message in the queue does not satisfy any case (while operating in SEQ mode), then the next message is tried. If none of the messages can be matched, a retry operation is performed (since a new message that matches may have arrived in the meantime). The other messages may be directed at RCV statements which are executed later on (and could be triggered by one of the current ON statements). When operating in the FIRST mode, if the first case can not be satisfied by any message in the queue, then the next case is tried, etc... If none of the messages match a case, a retry is performed.

The synchronous receive statement <SYNC RCV STMT> does not distinguish between foreign and local message classes, since it expects all the referenced message classes to be locally defined. Also, this statement has no queue processing options and no guards in the ON statements.

A synchronous module returns results and control to the calling module by means of a <RETURN STMT> (see explanation above). This differs from the asynchronous module, which responds to messages by means of SEND statements.

The entry receive statement, <ERCV STMT>, contains an optional message class part and a formal parameter part. This statement is used when only one of the entries in a module can be legally accessed from outside the module. Such a situation may arise when the module has only one entry, or alternatively, if it contains several entries, of which all but one are internal to the module and can not be accessed from outside it.

Notice that the ERCV statement is actually an abbreviation of the SRCV statement. Thus the following statements are equivalent:

```
ERCV  id1 (parm1, parm2, ... );
and
SRCV;
  ON  id1 (parm1, parm2, ... ) THEN EXIT;
  END;
```

The <REQUEST STMT> represents a composite form of communication. It requires all associated message classes to be foreign (i.e: defined by the called module) since they can be viewed as procedures in an abstract data type module. Notice that if an object contains only one entry point, it is necessary to enter the object name, followed by a dot.

The request statement can be used both for asynchronous and synchronous communication. When it is used between asynchronous machines, a dialog identifier (see explanation of <SEND STMT> above) is automatically attached to the message (if it is not already specified by the user). The receiver then appends the same dialog identifier to the response. This enables the system to recognize the response to a specific request among several messages in the queue.

When the request statement is used to communicate with a synchronous object, it passes the message and the machine to the object and waits until a response message is returned. The body of a request statement contains ON statements (like the synchronous receive statements). However in this case they check the message class of the **result** of the request.

The <INVOKE STMT> is a shorthand form of object reference, which creates an instance of the specified definition, makes a REQUEST of the instance and finally destroys the instance (after receiving the response). Like the <REQUEST STMT>, the ON part of an invoke statement checks the message class of the result of the invocation.

The following examples illustrate the usage of the above given constructs:

1.

```
SENDF  CREDIT.CHK (CUSTID)  VIA I+1 ;
 ....
MOD CREDIT PROCESS;
 ....
RCV ?J;
  ONL CHK (CUSTID) THEN DO;
    IF LIM (CUSTID) GT 0 THEN  SENDL OK (LIM)  VIA $DIAL(?J)
       ELSE  SENDL NOK  VIA $DIAL(?J);
    END;
  ONF TELLER.MOD (CUSTID, NEWLIM) THEN DO...END;
  END;
 ....
ENDMOD;
```

In this example the user checks the credit of a customer whose account number is CUSTID. This is done by sending message class CHK with argument CUSTID to object CREDIT. Notice that a SENDF statement is used, since the referenced message class is defined by CREDIT. The VIA part is used to identify the ongoing dialog by means of the current value of I+1.

The receive statement (in object CREDIT) accepts the message by the first ON statement, and its identifier is assigned to ?J. Then it looks for message class CHK. When it is found, the required credit check is performed, and the result is sent back to the caller. The $DIAL(?J) function in the VIA part provides the identifier of the dialog of which message ?J is part. Notice that in this case the message classes OK and NOK are defined locally to the object CREDIT, therefore a SENDL statement is used.

In order to enable this "conversation" to continue, RCV statements have to appear in the code of the sender, following the SENDF statement (but not necessarily immediately after it). These receive statements have to contain ONF statements to accept the results sent by the above described SENDL statements inside CREDIT.

2.

```
REQ CREDIT.CHK (CUSTID);
  ON OK (LIM) THEN GOTO CHKLIM;
  ON NOK THEN GOTO REFUSE;
  END;
```

Here the same credit check as in the above given example is performed, however in this case the user is oblivious to the kind of communication taking place (i.e: synchronous or asynchronous). As stated above, the associated message class has to be foreign (i.e: defined in CREDIT).

3.

```
RESULT= CREDIT2.CHK (CUSTID_ARG);
  ....
MOD CREDIT2 PROCEDURE;
  ....
SRCV;
  ON CHK (CUSTID) THEN DO;
    IF LIM (CUSTID) GT 0  THEN RETURN  (LIM)
    ELSE RETURN (0) REENTRY X;
    END;
  ON MOD (CUSTID, NEWLIM) THEN DO; ... ;END;
END;
X:SRCV;
  ....
ENDMOD;
```

This is the first example rewritten for synchronous communication. Of special interest are the usage of the RETURN statement (instead of the SEND statement in asynchronous communication) and the call to function CHK in object CREDIT2. The latter is actually an abbreviated REQUEST, in which CHK returns one value only (see explanation on shorthand requests and invocations, below). As stated above, the reentry part stipulates that next time module CREDIT2 will be called, execution will start at the statement labelled X.

PL/IDE also features abbreviated forms of requests and invocations, which are used when the expected response can be of one message class only. The following examples illustrate these options:

1. STACK.POP(//ELEMENT)
2. STACK.POP
3. COS%METHOD3(X//COSX)
4. PRINT(X)

The first example calls the object POP for an instance of the object STACK. The returned value is to be assigned to ELEMENT. In the second example the same call is made, but in functional reference form (i.e: the first example is a statement, whereas the second example is an expression whose value is the returned argument).

The third example is a shorthand invocation. It creates an instance of object definition COS, and sends argument X to entry point (message class) METHOD3. The returned value appears in COSX. As stated above after returning the result, the instance of COS%METHOD3 is destroyed. The last example is also an invocation. The entry specifier is not required, since PRINT has only one entry. If PRINT returns a result, the form represents a functional reference.

```
18   <STMT LIST> ::= <STATEMENT>
19                 | <STMT LIST> ; <STATEMENT>
20   <STATEMENT> ::= <UNLABELLED BASIC STMT>
21                 | <IDENTIFIER> : <STATEMENT>
22   <UNLABELLED BASIC STMT> ::= <BALANCED STMT>
23                             | <UNBALANCED STMT>
24   <BALANCED STMT> ::= <ASSIGNMENT STMT>
           | <ASSIGN STMT>
25         | <DO STMT>
26         | <FOR STMT>
27         | <WHILE UNTIL STMT>
28         | <CASE STMT>
29         | <GOTO STMT>
30         | <RETRY STMT>
31         | <EXIT STMT>
32         | <NEXT STMT>
33         | <RETURN STMT>
34         | <SEND STMT>
35         | <ASYNC RCV STMT>
36         | <SYNC RCV STMT>
           | <ERCV STMT>
37         | <REQUEST STMT>
38         | <INVOKE STMT>
           | <CANCEL STMT>
39         | <REFERENCE>
40         | <REFID> <SHORT ARGS>
41         | IF <EXPRESSION> THEN <BALANCED STMT> ELSE <BALANCED STMT>
42   <UNBALANCED STMT> ::= IF <EXPRESSION> THEN <UNLABELLED BASIC STMT>
43        | IF <EXPRESSION> THEN <BALANCED STMT> ELSE <UNBALANCED STMT>
44   <ASSIGNMENT STMT> ::= <TARGET LIST> <ASSIGNMENT OP> <COMPOUND EXP>
56   <ASSIGN STMT> ::= ASSIGN <ARG EXPRESSION>
45   <TARGET LIST> ::= <TARGET ELEMENT>
46                   | <TARGET LIST> , <TARGET ELEMENT>
47   <TARGET ELEMENT> ::= <REFERENCE>
48                     | <EXTRACTOR>
                       | <OBJECT ID>
                       | <DOT> <LEFT AGG BR> <RIGHT AGG BR>
                       | <IDENTIFIER> <DOT> <LEFT AGG BR> <RIGHT AGG BR>
49   <COMPOUND EXP> ::= <ARG EXPRESSION>
50                     | <COMPOSITE EXP>
50   <COMPOSITE EXP> ::= <QUESTION MARK> ; <STMT LIST> ; END
51   <DO STMT> ::= <TRANS> DO ; <STMT LIST> ; <END STMT>
52   <FOR STMT> ::= <TRANS> FOR <QUERY> <FOR CONT> ;
                                              <STMT LIST> ; <END STMT>
53   <FOR CONT> ::= <BY CLAUSE> <WHILE CLAUSE>
54                | <WHILE CLAUSE>
55   <BY CLAUSE> ::= BY <EXPRESSION>
56                 | <BY CLAUSE> BY <EXPRESSION>
57   <WHILE CLAUSE> ::= WHILE <EXPRESSION>
58                    |
59   <WHILE STMT> ::= <WHILEUNTIL> <EXPRESSION> ; <STMT LIST> ;
                                                        <END STMT>
     <WHILEUNTIL> ::= WHILE
```

```
                    | UNTIL
 60   <CASE STMT> ::= CASE <CONDITION LIST> ; <OF LIST> ; <END STMT>
 61                 | CASE <CONDITION LIST> ; <OF LIST> ;
                                    <OTHERWISE CLAUSE> ; <END STMT>
 62   <CONDITION LIST> ::= <EXPRESSION>
 63                      | <CONDITION LIST> , <EXPRESSION>
 64   <OF LIST> ::= <OF CLAUSE>
 65               | <OF LIST> ; <OF CLAUSE>
 66   <OF CLAUSE> ::= OF <COMPARISON LIST> THEN <UNLABELLED BASIC STMT>
 67   <COMPARISON LIST> ::= <COMPARISON ELEMENT>
 68                       | <COMPARISON LIST> , <COMPARISON ELEMENT>
 69   <COMPARISON ELEMENT> ::= <DONT CARE>
 70                          | <RELOP> <EXPRESSION>
 71                          | <SET RELOP> <EXPRESSION>
 72   <OTHERWISE CLAUSE> ::= OTHERWISE <UNLABELLED BASIC STMT>
 73   <RETRY STMT> ::= RETRY <OPTIONAL LBL>
 74   <EXIT STMT> ::= EXIT <OPTIONAL LBL>
 75   <NEXT STMT> ::= NEXT <OPTIONAL LBL>
 76   <GOTO STMT> ::= GO TO <IDENTIFIER>
 77                 | GOTO <IDENTIFIER>
 78   <END STMT> ::= END <OPTIONAL LBL>
 79   <OPTIONAL LBL> ::= <IDENTIFIER>
 80                    |
124   <TRANS> ::= TRANS <RESERVE PART>
125             | TRANS ONFAIL <IDENTIFIER> <RESERVE PART>
                |
      <RESERVE PART> ::=
                       | RESERVE ( <IDENTIFIER LIST> )
      <IDENTIFIER LIST> ::= <IDENTIFIER>
                          | <IDENTIFIER LIST> , <IDENTIFIER>
126   <CANCEL STMT> ::= CANCEL <IDENTIFIER>
127                   | CANCEL <IDENTIFIER> <GOTO STMT>
```

The <ASSIGNMENT STMT> contains the following parts:

- Target list - consists of one or more target elements separated by commas. A legal target element may be one of the following:

  - a <REFERENCE> - which represents a literal function,
  - an aggregate extractor - see explanation on EXTRACTORs above,
  - an <OBJECT ID> - which represents the object itself (as opposed to a pointer to the object),
  - an empty aggregate (.[]) - which represents all the local data, or
  - an identifier followed by an empty aggregate (id1.[]) - which represents the data local to the object referenced by the identifier.

- Assignment operator - which may be one of the following:

  - Simple assignment (=) - substitutes the contents of the left hand side with the result of the evaluation of the right hand side. This operator can be used with aggregate expressions, sets, tuples and, of course, scalars.

- Additive assignment (.+=) - this operation is valid for sets and aggregate expressions. For sets, it performs a union operation between the tuples resulting from the evaluation in the right hand side and the set specified in the left hand side (provided their types and dimensions match). Whereas for aggregates, this operation is repeated for all the sets in the aggregate on the left hand side, whose names match those of all the sets on the right hand side (see explanation of UNION in "usage of aggregate expressions", above).
- Substractive assignment (-=) - this operation is similar to the additive assignment, however in this case the evaluated tuples are deleted from the corresponding sets. If a tuple to be deleted does not exist in the set on the left hand side, the tuple is ignored. For aggregates, see explanation of MINUS in "usage of aggregate expressions".

- Compound expression - this may be any expression which is also a legal argument, or it may be a <COMPOSITE EXP>. This is a list of statements preceded by a question mark, and followed by "END". A composite expression is generally used when a variable is to receive different values, depending on the result of several conditional statements. Thus, in order to improve readability of the program, this variable is placed on the left hand side of an assignment statement, whose right hand side contains a composite expression. A value is assigned to the target variable by means of ASSIGN statements. The following example illustrates this point:

```
X= ?;
   IF A=B THEN ASSIGN (5)
   ELSE IF A<B THEN ASSIGN (8)
        ELSE ASSIGN (B);
   END;
```

  Notice that although the ASSIGN statements are considered <BALANCED STATEMENT>s, their usage is restricted to composite expressions.

FOR statements perform the statements in their statement list for each element in the set resulting from the calculation of the <QUERY> (see SETS and TUPLES above). The QUERY is restricted by optional WHILE and BY clauses. The WHILE clause has the same meaning as in other programming languages and requires no explanation. The BY clause specifies one or more attributes according to which the elements in the set are to be ordered (before performing the iteration). The first attribute in the BY clause is the dominant sorting factor, then all the elements with equal values for the first attribute are sorted BY the second attribute, etc...

The CASE statement is best viewed as a series of IF statements, in which the **common** left hand side of **corresponding** comparisons has been "factored out" and presented in the line preceding the first IF statement. The following example demonstrates this feature:

```
CASE    X ,  F(Y);
  OF  EQ 5, CTNS B  THEN  DO; ... ; END
  OF  GT 5,   *     THEN  Y= X + Z
  OTHERWISE  OBJ1.FUN2 (X,Y // B,C)
  END;
```

This example is equivalent to the following sequence of statements:

```
IF X EQ 5  AND F(Y) CTNS B  THEN  DO; ... ; END
ELSE IF X GT 5  THEN  Y= X + Z
     ELSE  OBJ1.FUN2 (X,Y // B,C)
```

Notice the use of the asterisk (*) where the value of the corresponding left hand side is of no consequence. In order to improve readability, the CASE statement uses OF and OTHERWISE instead of the keywords IF and ELSE respectively.

The <RETRY STMT> may be used only inside a CASE statement. When not accompanied by a label, control has to return to the **beginning** of the innermost CASE statement enclosing the RETRY statement. Otherwise, control has to return to the beginning of the enclosing CASE statement identified by this label.

The <EXIT STMT> may be used inside CASE, FOR, DO and RCV statements. When not accompanied by a label, control is transferred to the first statement **after** the compound statement. Otherwise, control is transferred to the first statement following the enclosing compound statement identified by this label.

The <NEXT STMT> may be used only inside FOR statements. When not accompanied by a label, it signals that the **next iteration** in the innermost enclosing FOR statement, is to be performed. Otherwise, the next iteration in the enclosing FOR statement identified by this label is executed.

The DO, GOTO, IF-THEN-ELSE and END statements are standard and require no explanation.

The <TRANS> part, which optionally prefixes the REQ, DO and FOR statements, is used to identify those statements as transactions. The format of the TRANS part is as follows:

```
TRANS  ONFAIL label  RESERVE ( id1,id2,... )
```

Where the ONFAIL part and the RESERVE part are optional. When TRANS is prefixed to a FOR statement, each iteration is a separate transaction. The ONFAIL part specifies the block of statements to which control is transferred when the transaction fails. If it is absent, enclosing transactions are backed out until either an ONFAIL clause is found or the enclosing process is destroyed. If the ONFAIL part is present, control is transferred to the specified label. If the target block is not at the same transaction level as the statements containing the ONFAIL clause, the executing program has to back out to the lowest common transaction level of the source and the target transaction.

The RESERVE part acts as a lock on the objects specified in the adjoining list. The reserved objects can not be accessed by another transaction until they are released by the current transaction (this happens when the transaction is terminated). Objects are usually reserved in order to write on their data. If an object is not reserved, the transaction can only access it on a READ-ONLY basis.

The <CANCEL STMT> has the following format:

```
CANCEL label1 GOTO label2
```

Where the GOTO part is optional. It causes the cancellation of the transaction which starts at label1. If the GOTO part is present, control is transferred to label2, thus overriding any prior directive (given either by default or by the ONFAIL part of the TRANS clause at the beginning of the transaction).

```
 1   <PROGRAM> ::= <MODULE> ; _|_
 2   <MODULE> ::= MOD <IDENTIFIER> <PROC> ; <PROC CONT>
 3              | MOD <IDENTIFIER> ONLINE ; <MOD CONT>
 4              | MOD <IDENTIFIER> <IDENTIFIER> TO <BASIC FORM> ;
                                                          <MOD CONT>
 5   <PROC> ::= PROCEDURE
 6            | PROCESS
 7   <BASIC FORM> ::= <PROC>
                    | ONLINE
 8   <PROC CONT> ::= <INIT BLK> <STMT LIST> ; ENDMOD
         | <DCL BLK> ; <INIT BLK> <STMT LIST> ; ENDMOD
 9       | <INIT BLK> <MOD LIST> ; <STMT LIST> ; ENDMOD
10       | <DCL BLK> ; <INIT BLK> <MOD LIST> ; <STMT LIST> ; ENDMOD
11       | <INIT BLK> <STMT LIST> ; <MOD LIST> ; ENDMOD
12       | <DCL BLK> ; <INIT BLK> <STMT LIST> ; <MOD LIST> ; ENDMOD
13       | <INIT BLK> <MOD LIST> ; <STMT LIST> ; <MOD LIST> ; ENDMOD
14       | <DCL BLK> ; <INIT BLK> <MOD LIST> ; <STMT LIST> ;
                                              <MOD LIST> ; ENDMOD
15   <MOD CONT> ::= <DCL BLK> ; <MOD LIST> ; ENDMOD
16                | <DCL BLK> ; ENDMOD
                  | <MOD LIST> ; ENDMOD
                  | ENDMOD
17   <DCL BLK> ::= DCL ; <BASICAGG> ; ENDDCL
18   <INIT BLK> ::= INIT ; <STMT LIST> ; ENDINIT ;
17                |
18   <MOD LIST> ::= <MODULE>
19                | <MOD LIST> ; <MODULE>
```

A program in PL/IDE is a module, which may contain other modules. There are three basic system defined modules: PROCESS, PROCEDURE and ONLINE. The language also provides the user with the capability to define other kinds of modules in terms of these basic modules.

The declaration block, <DCL BLK> of a module has the structure of a basic aggregate constructor, in which the relation names are keywords that specify entries, exits, entities, relationships, etc... (A detailed description of the <DCL BLK> is presented elsewhere).

The initialization block, <INIT BLK>, may appear only in PROCEDUREs and PROCESSes.

A description of the structure of these modules follows. A detailed explanation of their usage is given elsewhere.

PROCESSes and PROCEDUREs have the same structure:

```
MOD id1 PROCEDURE ;
   <DCL BLK> ;
   <INIT BLK> ;
   <MOD LIST> ;
   <STMT LIST> ;
   <MOD LIST> ;
   ENDMOD ;
```

Where a module list is consists of one or more "in place" module definitions, and both module lists are optional.

PROCEDUREs are used for synchronous communication, whereas PROCESSes are used for asynchronous communication. Both may have several entry points, and the data defined in the <DCL BLK> can not be accessed from outside.

As stated above, other modules may be defined in the language. These modules are translated into one of the three basic types prior to compilation. The target module is specified in the TO-part of the module header.

The language features two kinds of modules, for which it provides a "translator". These special modules are DATA GROUPs and ABSTRACTIONs. Both have the following structure:

```
MOD id2 DATA_GROUP  TO  PROCESS   ;
        ABSTRACTION     PROCEDURE
   <DCL BLK> ;
   <MOD LIST> ;
   ENDMOD ;
```

Where the module list and the declaration block are optional for DATA GROUPs and required for ABSTRACTIONs. DATA GROUPs may also be translated into PROCESSes, whereas ABSTRACTIONs can only be of procedural type.

DATA GROUP procedures and ABSTRACTIONs are used for synchronous communication, whereas the DATA GROUP process is used for asynchronous communication. Like PROCEDUREs and PROCESSes, DATA GROUPs and ABSTRACTIONs may have several entry points, however unlike PROCEDUREs and PROCESSes, they contain no code. In DATA GROUPs the data defined in the <DCL BLK> can be accessed from outside the module. This can be accomplished by means of an expression like:

```
id2.name3 ,
```

where id2 is the identifier of the module. Whereas for ABSTRACTIONs the declared data is local, and thus inaccessible from outside the module.

The user defined modules enable the user to specify new applications. In order to do this, the user has to supply a partial compiler, that translates the modules in the new application (consisting only of declarations) to any of the **three** basic modules. The following example will illustrate this concept:

```
MOD id3 REPORT_GENERATOR TO PROCEDURE;
   DCL;
     ...
     END;
   MOD header;
     ...
     ENDMOD;
   MOD table_of_contents;
     ...
     ENDMOD;
   ENDMOD;
```

In this example the user provides a compiler for REPORT_GENERATOR. This compiler translates all the declarations in id3 to a module of type PROCEDURE. In this context, user defined modules have to be aware of the implementation of the declared variables.

The following table summarizes the above explained concepts:

module kind	comm	access to	user code
PROCEDURE	sync	user defined entries	yes
PROCESS	async	user defined entries	yes
DATA GROUP procedure	sync	user defined entries declared sets	no
DATA GROUP process	async	user defined entries declared sets	no
ABSTRACTION	sync	user defined entries	no
user defined	both	?	no