THE HIERARCHICAL LANGUAGE SYSTEM

Nobuyoshi Terashima

Yokosuka Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation

1-2356 Take, Yokosuka-shi 238, Japan

Phone No. (0468) 59-2716

ABSTRACT

The Hierarchical Language System HLS is presented. HLS consists of the structured programming language SPL and the extensible programming language EPL which is built upon SPL layer and provides statement and expression extention facilities, and provides to automate top down programming as much as possible by the following means.

(1) To describe each breakdown process in the programming languages EPL and SPL, in order to eliminate ambiguity of each breakdown representations which are in general written in natural languages.

(2) To generate the whole system which runs in the actual programming environment by this system.

This paper describes HLS concepts and language specifications of its components.

HLS effectiveness is to be presented in the near future.

1. INTRODUCTION

It is assumed that software systems be classified into the following fields; operating systems, compilers, utilities and application programs.

In each field, there are many routines, which are characteristic of the field and occur in the field frequently. When using PL/I, COBOL or an assembler language to write those routines, they are described as macros or (function) subroutines, and invoked by means of macro or subroutine references when necessary for processing.

This invocation method cannot contribute to enhancement of programming readability and descriptive convenience.

Therefore, it is necessary that readability and descriptive convenience should be enhanced by providing a descriptive means fitted for the field, because it is expected that readability and descriptive convenience might contribute to enhancement of programming reliability and productivity as well as program maintenance.

The following implementations are enumerated to satisfy the above requirement:

(1) To develop a new language fitted for a particular field.

(2) To make use of extensible programming language concepts [21, 22, 23].

The former implementation requires development of a new compiler language for each field. Therefore, in setting up and maintenance the compiler language, a very large cost is involved. Therefore, in this research, the latter implementation is taken into consideration.

Generally speaking, an extensible programming language has a base language besides extension facilities. Therefore, the hierarchical language system HLS is proposed, which consists of the structured programming language SPL as the base language and the extensible programming language EPL, which is built upon SPL layer. It is expected that HLS might contribute to enhancement of programming reliability and productivity as well as program maintenance by the following reasons.

(1) Programming reliability and productivity as well as program maintenance can be enhanced by SPL.

Effectivity can be enhanced by EPL. Effectivity might be one of the major contributing factors for enhancement of programming reliability and productivity as well as program maintenance.

(2) HLS can provide a means by which representations, which are written in EPL and suitable to a particular field, are broken down into representations, which are written in SPL and transparent to the field, in a top down manner.

This paper describes HLS concepts, language specifications of HLS components.
Especially, SPL and EPL language specifications are described in detail, because they are new programming languages proposed in this paper.

2. HLS CONCEPTS

HLS is the two level hierarchical language system, which consists of SPL as the base language, and EPL, which is built upon SPL layer and provides statement and expression extension facilities.

The reasons why EPL does not have extension facilities of language constructs, other than statements and expressions, are as follows:

(1) References of macros and (function) subroutines can be easily represented by new statements or expressions using EPL.

(2) Program structure, data declarations and data references should be made well structured in order to enhance programming reliability and productivity, hence these constructs should be the same as those of SPL.

The basic concept of top down programming is that programs consist of layers, each layer implementing a characteristic level of abstraction built upon the layers below and supporting the layers above [1, 2, 3, 5, 6, 7, 8, 12, 14, 15, 16, 17, 19]. In HLS system, programs of a field

consist of layers which should be described in the representations characteristic of the field using EPL and layers which should be described in the representations transparent to the field using SPL.

In other words, the entire system operation is described in the representations fitted for a field using EPL. They are then broken down into the layers below.

This process continues until everything is resolved into procedures which are written in SPL, described in the representations transparent to the field and supported in the implementation environment.

This breakdown procedure is represented in a multi-layer structure, as follows:

$$\begin{array}{l} \overline{L_{(1)AP}} \\ L_{(2)AP} \\ \vdots \\ \overline{L_{(n)AP}} \\ L_{(n+1)NAP} \\ L_{(n+2)NAP} \\ \vdots \\ \overline{L_{(n+m)NAP}} \end{array}$$

where (1) $L_{(1)AP\sim(n)AP}$ are layers which should be described in the representations fitted for a particular field and are written in EPL.

(2) $L_{(n+1)NAP\sim(n+m)NAP}$ are layers which should be described in representations transparent to the field and are written in SPL.

The set of procedures in a layer $L_{(i)}$ is written as $\{P_{L(i)}\}$. The set with no procedures should not be allowable. The entire system operation of a field is described as $\{P_{L(1)}\}$. It is then broken down into $\{P_{L(2)}\}$. This process continues until everything is resolved into $\{P_{L(n+m)}\}$.

Breakdown of a layer into the next layer below is described using EPL and SPL. Namely, in EPL, a layer is described as the syntax description and the next layer below is descrived as the semantic description.

In SPL, a layer is described as procedures, which contain the subroutine references, and the next layer below is described as the subroutines.

Breakdown is performed as follows:

(1) Breakdown of layer $L_{(k)AP}$ into layer $L_{(k+1)AP}$ as well as breakdown of layer $L_{(n)AP}$ into layer $L_{(n+1)NAP}$ are performed using EPL, where k = 1, 2, ..., n-1.

(2) Breakdown of layer $L_{(n+j)NAP}$ is performed using SPL, where j = 1, 2, ..., m-1.

This is illustrated using the example shown in Section 4.1 (2).

(1) The construct FOR A = B TO C, D BY E TO F, G TO H BY I statement is a representation of layer $L_{(1)AP}$, which is characteristic of of a field and written in EPL, and a construct which is contained in a procedure of layer $L_{(1)AP}$.

(2) The statement list DO A = B TO C; <statement> ENDDO;
DO A = D BY E TO F; <statement> ENDDO;
DO A = G TO H BY I; <statement> ENDDO;
is a representation of layer $L_{(2)NAP}$, which is transparent to the field and written in SPL, and a procedure of layer $L_{(2)NAP}$.

(3) The construct contained in the procedure of layer $L_{(1)AP}$ is implemented by the procedure of layer $L_{(2)NAP}$. This illustrates that layer $L_{(1)AP}$ is broken down into layer $L_{(2)NAP}$, using EPL.

SPL belongs to a class of languages such as PL/I[25], does not have a GO TO statement [4, 11, 18], has salient control structures such as GCASE statement, in addition to the control structures which a structured programming language has in general [5, 6, 8, 20], and has inline machine code facilities for use of basic system description [1, 9, 10].

In SPL, data declarations and references are made well structured [13].

Data scope includes global data scope and local data scope. Global data may be used in the local blocks contained in a block where the global data are declared.

On the other hand, local data should be available only in a block where the local data are declared, but they should not be used in the local blocks contained in a block where the local data are declared.

This shows that data design procedure should be performed from the outermost block to the innermost block and that this concept may contribute to top down programming strategy.

EPL is the extensible language whose base language is SPL and has extension capabilities of statements or expressions [21 ∿ 24].

One of EPL characteristic features is that syntax and semantic description may be performed hierarchically, using the structured descriptive method.

It is expected that this might support the powerful descriptive means for extension.

3. SPL LANGUAGE SPECIFICATIONS

3.1 Basic Concepts

(1) SPL is one of the block structured languages and has data types and structures, such as PL/I [25].

(2) Procedure consists of the declaration segment followed by the procedure body.

(3) Declaration segment consists of entry declaration section, global declaration section and local declaration section.
When the declaration of an identifier is made in a block, there is a certain well-defined region of the program over which this declaration is applicable.
This region is called the scope of the declaration or the scope of the name established by the declaration.
Data declarations in the local section are applicable only in the block where they are declared. Parameters and the entry data should be declared in the local section or the global section.
Data declarations in the global section are applicable in the block as well as the blocks contained in it.
Data declarations in the entry section can be used to communicate between different external procedures. Only the entry data is declared in this section.

(4) SPL has GO TO-free control structures as follows:

(a) Generalized CASE statement

(b) IF statement (selection of a statement based on the testing of a condition).

(c) REPEAT, WHILE DO and DO statements (iteration).

(d) EXIT statement in a loop (exit mechanism from loop).

(e) An internal procedure.

(f) BEGIN block.

BEGIN block is the same as that of PL/I.

(g) A machine code block where machine instructions can be used.

The statements which control structures link together include assignment statements, calls of other procedures, machine instructions in the machine code block, the RETURN statement in a procedure, the storage allocation or freeing statement, and the input/output statements.

(5) In-line machine code facilities

This facility permits the user to retain control over machine capabilities which are needed to perform basic system functions.

3.2 SPL Characteristic Features

SPL characteristic features related to control structures are described in this section.

SPL specifications other than the above features are not shown in this paper.

The symbols < and > are used to enclose the name of a general syntactic entity. Braces ({ }) are used for grouping and specify the occurrence of one of the group items.

The vertical stroke indicates that a choice is to be made. The square brackets [ ] denote options.

Three dots ... denote the occurrence of the immediately preceding syntatic entity one or more times in succession. For example, the following statement is given:

CALL <identifier> [(<parameter-list>)];

This means that this statement consists of the keyword CALL followed by an identifier. The identifier may optionally be followed by a parameter list, enclosed in parentheses. The CALL statement is terminated by a semicolon.

(1) Data Types and Organization

Data types include arithmetic (binary/decimal, fixed point/floating point) data, string (character/bit) data, locator (pointer/offset) data and area data [25]. File names, entry names and condition names are not considered to be data.

Data organization includes scalar data items, aggregates of data items (arrays and structures). A data item may be either a constant or the value of a scalar variable.

Constants and scalar variables are called scalar items. All classes of variable data items may be grouped into arrays or structures as PL/I [25].

(2) A program is composed of more than or equal to one external procedure.

(3) Procedure

There are two types of procedures; external procedures and internal procedures.

The general form of a procedure is given as follows:

<procedures> : = <procedure-beginning-part> <data-declaration-segment><procedure-body><procedure-ending-part>.

General form of <procedure-beginning-part> is given as follows:

<procedure-beginning-part> : = <identifier> : PROC [(<parameter-list>)] [RETURNS (<attribute>)];

where <identifier> is the name of the procedure. Each item of <parameter-list> is delimited by a comma, and is an arithmetic data item, a string data item, a locator data item or an area data item.

The data item should be declared in the <local-declaration-section>of the <data-declaration-segment>.

The RETURN option should be specified when the procedure is a function procedure.

Data variables, which are used in the <procedure-body>, should be declared in the <data-declaration-segment>. The <data-declaration-segment> consists of <entry-declaration-section>, <global-declaration-section> and/or <local-declaration-section>. It is delimited by the DATA-SEGMENT statement and the END-SEGMENT statement.

An invoked external procedure name is declared in the <entry-declaration-section>. It is delimited by the ENTRY-SECTION statement and the END-SECT statement. Data which is accessible in a block and the blocks contained in it, is declared in the <global-declaration-section>.

The global data should be declared in each block where it is used. It is delimited by the GLOBAL-SECTION statement and the END-SECT statement.

The local data should be declared only in the block where it is used. It is delimited by the LOCAL-SECTION statement and the END-SECT statement.

When it is desired to use an identifier as a different entity in the encompassed block, it is necessary to specify the local attribute to it in the encompassing block.

The identifier declared as a global data should not be specified for another purpose in all of the procedures which are contained in an external procedure.

The <procedure-body> is constructed from GO TO-free control structures and/or statements described in Section 3.1 (4), except the constructs which consist of only the internal procedures.

The <procedure-ending-part> is the END statement.

The syntax rule is given by

END <identifier> ;

where <identifier> is the same identifier as the procedure name of the <procedure-beginning-part>.

(4) Expressions

(a) General Forms

Expression are scalar expressions. A scalar expression consists of a constant, a scalar variable, a scalar expression enclosed in parentheses, two scalar expressions connected by an infix operator, or a function reference that returns a scalar value.

Syntax rule of a scalar expression is shown in Fig. 1.

The priority of operations is shown, from highest to lowest, as follows:

(1) **, ¬
(2) *, /
(3) +, -
(4) <=, <, ¬<, ¬=, =, >=, >, ¬>
(5) &
(6) !

Operations within an expression are performed in the order of decreasing priority.

If an expression is enclosed in parentheses, it is treated as a single operand and evaluated first.

Within nested parentheses, evaluation proceeds from the least inclusive set to the most inclusive set.

When operations with the same priority occur in an expression, evaluation is performed from left to right.

I of <expression i> corresponds to the priority of the operation where <expression i> appears.

The smaller the value of i, the higher the priority of the operation.

In this paper, a scalar expression, which consists of a single operand without an operator and is not a function reference nor a built-in function, is called a scalar variable.

The attributes of all operands of a scalar expression should be the same.

(b) Operators

(i) Operands of bit string operations such as '¬(not)', '& (and)' and '! (or)' operations, are bit string expressions.

(ii) Operands of comparison operations, such as '<=', '<', '=', '>', '>=', and '¬=' operations, are arithmetic expressions, bit string expressions or character string expressions.

Operands of operations, such as '=' and '¬=', are locator expressions.

1. Bit string comparison, which involves the left-to right comparison of binary digits. If the strings are different lengths, the shorter is extended on the right with zeros. Comparison of bit string operations is performed on a bit-by-bit basis. When the values of the corresponding bit positions of the operands are different, then it is assumed '1'B is greater than '0'B, and the comparison result is obtained. The '=(equal to)' operation stands when all of the values of the corresponding bit positions of the two operands are the same. The result of a comparison is a bit string of length one. The value is '1'B if the relationship is true or '0'B if it is false.

2. Character comparison, which involves left-to-right, pair-by-pair comparisons of characters, according to the implementation-defind collating sequence.

If the operands are of different lengths, the shorter is extended to the right with blanks.

(iii) Operands of the arithmetic operations, such as '+', '-', '*', '/', and '**' operations, are arithmetic expressions.

(5) Control Structures

Control structures are described in this Section. The syntactic unit <statement list> is constructed from the control structures and/or statements described in Section 3.1 (4), except for constructs which consist of only the internal procedures.

(a) IF Statement

The IF statement specifies evaluation of an expression and a consequent flow of control dependent upon the value of the bit string.

The syntax of the IF statement is given by IF <scalar-expression> THEN $\langle$statement-list$\rangle_1$ [ELSE $\langle$statement-list$\rangle_2$] ENDIF;

where the scalar expression is an expression whose result is a bit string of length one.

If the value is '1'B, the $\langle$statement-list$\rangle_1$ is executed and control is transferred to the statement following ENDIF.

If the value is '0'B, the $\langle$statement-list$\rangle_2$ is executed when it is specified, and control is transfered to the statement following ENDIF.

(b) Generalized CASE (GCASE) Statement

Three control consrtucts; SELECT CASE OF, SELECT FIRST ACTION and SELECT EVERY ACTION have been proposed to reduce complexity of nested IF constructs [26].

In actual situations, these control structures and extended control structures of them are used in a program context concurrently rather than independently.

Therefore, a control construct, which contains all of these three control structures and extended control structures of them, is required to reduce complexity of programs.

The generalized CASE construct is proposed to satisfy this requirement.

The syntax is given by

```
    GCASE;
(<expression_1>): <statement list>_1 [NEXT]
                  ELSE <statement list>_1e
(<expression_i>): <statement list>_i [NEXT]
                  ELSE <statement list>_ie
(<expression_n>): <statement list>_n [NEXT]
                  ELSE <statement list>_ne
            [COMMON <statement list>_c]
    ENDCASE;
```

where each <expression$_i$> is a scalar expression whose result is a bit string of length one.

If the value of <expression$_1$> is true, then <statement list$_1$> is executed and one of the following actions is taken:

(i) When the NEXT option is omitted from the specification and the COMMON part appears, the COMMON part (<statement list$_c$>) is executed and the other statement lists are not executed.

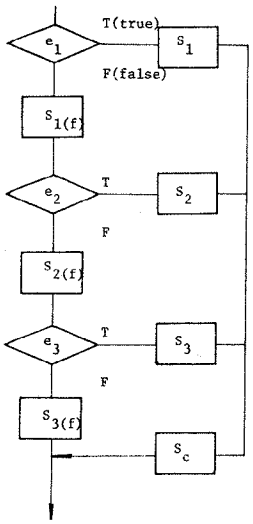
(ii) When the NEXT option is specified, the execution proceeds with the next construct (<expression$_2$>): <statement list$_2$> ELSE <statement list$_{2e}$>.

(iii) When both the NEXT option and the COMMON part are omitted from the specification, the execution proceeds with the statement following ENDCASE.

If the value of <expression$_1$> is false, the ELSE part (<statement list$_{1e}$>) is executed, and then the execution proceeds with the next construct (<expression$_2$>): <statement list$_2$> ELSE <statement list$_{2e}$>.

Then the similar actions are taken in case of the construct (<expression$_2$>): <statement list$_2$> ELSE <statement list$_{2e}$>. Thus, when there is no construct to be executed, the execution proceeds with the statement following ENDCASE.

The GCASE statement function is clearly illustrated by the following example.



where e_1, e_2 and e_3 are scalar expressions, and S_1, S_2, S_3, $S_{1(f)}$, $S_{2(f)}$, $S_{3(f)}$ and S_c are statement lists.

When $S_{1(f)}$ and $S_{2(f)}$ consist of statements other than null statements, this control structure can not be described by SELECT CASE OF construct.

Therefore it should be described, using nested IF constructs, as follows.

```
IF e1 THEN S1  Sc
     ELSE S1(f)
          IF e2 THEN S2
                     Sc
               ELSE S2(f)
                    IF e3 THEN S3
                              Sc
                         ELSE S3(f)
                              ENDIF;
          ENDIF;
ENDIF;
```

However, this control structure may be concisely described by the following GCASE construct.

```
          GCASE;
(e1):     S1
          ELSE S1(f)
(e2):     S2
          ELSE S2(f)
(e3):     S3
          ELSE S3(f)
          COMMON Sc
          ENDCASE;
```

This example shows one of the favorable capabilities of GCASE statement. The GCASE statement contributes to enhancement of understandability of a program where this type of control structure is used.

SELECT CASE OF, SELECT FIRST ACTION and SELECT EVERY ACTION [26] can be described, using GCASE statement, as follows.

(i) SELECT CASE OF Construct

A SELECT CASE OF construct example is given:

```
SELECT CASE OF TRANSACTION-CODE =
WHEN ('A') S1
WHEN ('D' OR 'X') S2
WHEN ('C') S3
WHEN NONE ARE SELECTED S4
WHEN ONE IS SELECTED Sc
ENDSELECT
```

where S_1, S_2, S_3, S_4 and S_c are statement lists.

This example is described, using GCASE statement, as follows.

```
     GCASE;
(TRANSACTION-CODE = 'A'):  S1
                           ELSE;
(TRANSACTION-CODE = 'D'!  TRANSACTION-CODE = 'S'):  S2
                                                    ELSE;
(TRANSACTION-CODE = 'C'):  S3
                           ELSE S4
                           COMMON Sc
 ENDCASE;
```

(ii) SELECT FIRST ACTION Construct

A SELECT FIRST ACTION construct example is given:

```
SELECT FIRST ACTION
```

WHEN RANK IS LESS THAN .10

S_1

WHEN RANK IS LESS THAN .50

S_2

WHEN RANK IS LESS THAN 1.00

S_3

WHEN NONE ARE SELECTED

S_4

ENDSELECT

where S_1, S_2, S_3 and S_4 are statement lists.

This example is described, using GCASE statement, as follows.

```
GCASE;
(RANK < 0.10):  S1
                ELSE;
(RANK < 0.50):  S2
                ELSE;
(RANK < 1.00):  S3
                ELSE S4
ENDCASE;
```

(iii) SELECT EVERY ACTION Construct

A SELECT EVERY ACTION construct example is given:

```
SELECT EVERY ACTION
WHEN ACE-IN-SUIT
        S1
WHEN KING-IN-SUIT
        S2
WHEN QUEEN-IN-SUIT
        S3
WHEN JACK-IN-SUIT
        S4
 ENDSELECT
```

where S_1, S_2, S_3 and S_4 are statement lists.

This example is described, using GCASE statement, as follows.

```
GCASE
(ACE-IN-SUIT):    S1 NEXT
                  ELSE;
(KING-IN-SUIT):   S2 NEXT
                  ELSE;
(QUEEN-IN-SUIT):  S3 NEXT
                  ELSE;
(JACK-IN-SUIT):   S4
                  ELSE;
ENDCASE;
```

(c) Iteration Functions

The following statements are allowed in SPL. <loop>: =

{[<exit-designator>:] <repeat-statement> <statement-list>

[<exit-block>] ENDREP;

|[<exit-designator>:] <do-statement$_1$> <statement-list>

[<exit-block>] ENDDO;

|[<exit-designator>:] <do-statement$_2$> <statement-list>

[<exit-block>] ENDDO;}

<repeat-statement>: = REPEAT { WHILE (<scalar-expression>) | FOREVER } ;

<do-statement$_1$>: = WHILE (<scalar-expression>) DO;

<do-statement$_2$> : = DO <scalar-arithmetic-expression> = <arithmetic-expression$_1$> {TO <arithmetic-expression$_2$> [BY <arithmetic-expression$_3$>] | BY <arithmetic-expression$_3$> [TO <arithmetic-expression$_2$>] } ;

<exit-block>: = EXITBLOCK; <exit-group> [<exit-group>] ...

<exit-group>: = (<exit-identifier>): <statement-list>.

(i) <repeat-statement> specifies that <statement-list>, immediately following, is executed once, and then the following action is taken:

① IF the FOREVER option is specified, the <statement-list>, immediately following, are executed endlessly. Therefore, this loop should be terminated by an EXIT statement.

② When the WHILE option is specified, the action is as follows:

If the value of <scalar-expression> is true ('1'B), then <statement-list> is executed. Otherwise, <statement-list> is skipped and execution proceeds with the statement following ENDREP. However, if <statement-list> is executed, then execution proceeds with the REPEAT statement again.

Thus, if <scalar-expression> is true, then <statement-list> is executed until <scalar-expression> become false ('0'B).

(ii) When the <do-statement$_1$> is specified, the action is equivalent to the preceding (i) ②

(iii) The effect of the <do-statement$_2$> would be the same as the DO statement (option 3) of PL/I.

(d) EXIT Statement

The synatx is given by

EXIT [<exit-designator>] [(<exit-identifier>)];

The EXIT statement may be specified in the <loop>.

(i) If <exit-identifier> is specified, then the statement list of the <exit-group> designated by it is executed and one of the following actions is taken:

1. When the <exit-designator> appears, the loop is terminated according to the <exit-designator>.
2. When the <exit-designator> is omitted from the specification, the loop where the EXIT statement is immediately contained is terminated.

(ii) If <exit-identifier> is not specified, one of the following actions is taken:

1. When the <exit-designator> appears, the loop is terminated according to the <exit-designator>.
2. When the <exit-designator> is omitted from the specification, the loop where the EXIT statement is

immediately contained is terminated.

(iii) When the <exit-designator> is specified, control is transfered to the statement following the ENDREP or ENDDO of the loop designated by it.

(iv) The function is illustrated by the following example.

```
LOOP 1:   REPEAT ...;
            :
LOOP 2:   REPEAT ...;
            :
LOOP 3:   REPEAT ...;
            :
          EXIT;          ... (1-1)
            :
          EXIT LOOP 2;   ... (1-2)
            :
          ENDREP;        ... (1)
            :
          ENDREP;        ... (2)
            :
          ENDREP;        ... (3)
```

When the EXIT statement (1-1) is executed, the execution proceeds with the statement following ENDREP (1). However, when the EXIT statement (1-2) is executed, the execution proceeds with the statement following ENDREP (2)

(6) Interrupt Handling

In PL/I [25], interrupt handling is manipulated, independently of sequential control flow. When an area condition or end-of-file condition occurs, the currently active ON statement is executed. That is, control is transfered non-sequentially. This sort of non-sequential control transference disrupts the well-structuredness of program control flow.

In order to make program control flow well-structured, the on-unit should be specified in the statement where an interrupt may occur.

For example, in case of the ALLOCATE statement,<the on-unit>is specified, as follows:

ALLOCATE <based-variable> SET (<locator-variable>) IN (<area-variable>) <on-clause>;

<on-clause>: = ON (<condition-name>) <statement-list> where the <condition-name> is AREA. The <on-clause> is used to specify the action when an area condition may occur. Similarly, the <on-clause> is also specified in the input/output statements.

(7) In-line Machine Code Facility

SPL allows machine instructions to be inserted into SPL programs, where they are contained in the code blocks. However, certain instructions will never be supported affecting control flow, such as a branch instruction which is equivalent to a GO TO statement. SPL variables may be used as operands of machine instructions. Compiler checks that the machine instruction is one of the allowable ones, and that operands are suitable for the instruction.

3.3 SPL Program Example

An example [8] is shown in Fig. 2.

4. EPL LANGUAGE SPECIFICATIONS

SPL and EPL are based on the concept of levels of abstraction [3]. EPL is the outer layer of SPL and provides the facilities to add new statements or expressions to the base language SPL.

Concretely speaking, using the EPL extension facilities, a new statement and expression can be added as constructs of the <procedure-body> and <scalar expressions> of SPL, respectively.

The new statement or expression, which has been defined by statement or expression syntax rule description, is interpreted, using the corresponding statement or expression semantic description. SPL source program are generated accordingly.

4.1 Statement Extension

Statement syntax to be extended is illustrated by

<statement>: = {<syntactic-unit>} ...

<syntactic-unit>: = {<parameter> | <character-string-constant>}

where <character-string-constant> specifies a statement keyword or a delimiter.

(1) Statement Extension Method

The syntax is given by

<statement-syntax-rule-description>

: = 1 <identifier>$_1$ SYNTAX-OF-STATEMENT

{, 2 <minor-structure>$_{1ss}$ [, 3 <minor-structure>$_{2ss}$ [, ...

[, n+1 <minor-structure>$_{nss}$] ...] ...] ... } ...;

<minor-structure>$_{1ss\text{-}nss}$: = { <character-string-constant>$_1$

| ITERATION (<parameter>$_2$) | DISORDER | OPTIONAL (<parameter>$_3$)

| <character-string-constant>$_2$ DEL | ONE-OF-THEM | <identifier>$_2$

| <parameter>$_1$ { VARIABLE / EXPRESSION i (i = 1, 2, ...) / STATEMENT } }

where a preceding number of <minor-structure> specifies a level number. A minor structure at level n contains all following items declared with level numbers greater than n up to but not including the next item with a level number less than or equal to n.

Elements of a minor structure at level n are defined as minor structures with level number n+1, which is contained in the minor structure at level n.

The <character-string-constant> specifies a keyword or delimiter of a statement to be extended.

The <identifier>$_1$ specifies the name which identifies the statement to be extended.

ITERATION (...) specifies that the contained elements be iterated n times, where n is the value of <parameter>$_2$.

DISORDER specifies that the occurrence order of the contained elements be disorderly.

OPTIONAL (...) specifies whether the contained elements exist or not, depending on the existence of <parameter>$_3$.

The <character-string-constant>$_2$ DEL may be one of the elements of a minor structure ITERATION (...), and show that <character-string-constant>$_2$ is generated on every iteration except the last one.

It is illustrated by the following example:

:

2 ITERATION ($1)

3 'ABC',

3 ',' DEL,

:

In this case, when the value of $1 is 3, the string 'ABC, ABC, ABC' is generated.

ONE-OF-THEM specifies that one of the contained elements is selected. The <identifier>$_2$ specifies the name which identifies a minor structure.

VARIABLE specifies that <parameter>$_1$ has the syntax type of a variable which is allowable in SPL specifications.

EXPRESSION i (i = 1, 2, ...) and STATEMENT specify that <parameter>$_1$'s have the syntax types of expression i (i = 1, 2, ...) and statement, respectively, which are allowable in SPL specifications or extended syntax.

(2) Statement Semantic Description

Two types of statement semantic description are provided in EPL.

(a) Type 1

The syntax is given by

<statement-semantic-description>

::= 1<identifier> SEMANTICS-OF-STATEMENT (<identifier>$_1$)
{, 2<minor-structure>$_{1sm}$ [, 3<minor-structure>$_{2sm}$ [..[, n+1 <minor-structure>$_{nsm}$]...]...]...}...;

<minor-structure>$_{1sm-nsm}$ ::={<character-string-constant> | <parameter>$_1$ | ITERATION (<parameter>$_2$) | OPTIONAL ({<parameter>$_3$ | ┐<parameter>$_3$})}, where the <character-string-constant> specifies a keyword or delimiter of a SPL statement, or of a new statement extended by EPL.

The <identifier> is the name of the statement semantics.

The <identifier>$_1$ is used to identify the corresponding <statement-syntax-rule-description>. When the semantics, which have already been defined, are used, only the element with level one should be specified.

The <parameter>$_1$, <parameter>$_2$ and <parameter>$_3$ are information passed from the <statement-syntax-rule-description> and have the same attributes or forms as those of the <statement-syntax-rule-description>.

Parameter passing rule between syntax definition and semantic definition is described in Section 4.3.

ITERATION (...) is the same as described in the preceding section.

OPTIONAL (<parameter>$_3$) specifies that contained elements exist, if <parameter>$_3$ exists.

OPTIONAL (┐<parameter>$_3$) specifies that contained elements exist, unless <paramecer>$_3$ exists.

The statement syntax extension example is shown in Fig.3.

In Fig. 3, when a source statement written in EPL is given as follows.

FOR A = B TO C, D BY E TO F, G TO H BY I

<statement>,

the following SPL statements are generated:

DO A = B TO C;
<statement>
ENDDO;
DO A = D BY E TO F;
<statement>
ENDDO;
DO A = G TO H BY I;
<statement>
ENDDO;

This example shows that SPL statements can be described compactly and conveniently using EPL.

(b) Type 2

In this type, semantics are described by a procedure. This is explained in the next Section 4.2 (2).

4.2 Expression Extension

Expression form to be extended is defined as follows:

<expression> ::= <expression>$_1$ <operator> <expression>$_1$ [<operator> <expression>$_1$] ...

where <expression>$_1$ is one of the SPL scalar expressions or one of the extended expressions.

Each <operator> is an i-th element O_i of an n-tuple operator (O_1, O_2, .. O_i, ... O_n), which represents a single operation. Each <operator> is described by a <character-string-constant>.

(1) Expression extension method

The syntax is given by

<expression-syntax-rule-description> ::= 1<identifier>$_1$
SYNTAX-OF-EXPRESSION j(j = 2, 3, ...) 2 <parameter>
EXPRESSION i(i = 1, 2, ...)
, 2 <character-string-constant>
, 2 <parameter> EXPRESSION i (i = 1, 2, ...)
[, 2 <character-string-constant>
, 2 <parameter> EXPRESSION i (i = 1, 2, ...)] ...

where the <identifier>$_1$ is the name of the expression syntax.

J of SYNTAX-OF-EXPRESSION j (j = 2, 3, ...) specifies the priority of the new operation.

EXPRESSION i is the same as that of <statement-syntax-rule-description>. Each <character-string-constant> specifies an i-th operator constituent O_i of an n-tuple operator (O_1, O_2, ... O_i, ... O_n) of an expression to be extended.

A new operation can be defined by using <expression i>. By this definition, the priority of the new operation is also given uniquely.

When it is desired to change an operation priority which has already been defined, operation definition set should be changed. This is illustrated by the following examples.

(a) When a new operator ADD is defined by <expression 4> ::= <expression 3> ADD <expression 3>, the old expression

definition <expression 4> : = <expression 3> [{ + 1 - }
<expression 3>] ... and the new definition <expression 4>:
= <expression 3>
ADD <expression 3> are merged into the following definition

$$\text{<expression 4>} : = \text{<expression 3>} \left\{ \begin{array}{l} \text{ADD <expression 3>} \\ \text{[\{ + 1 <expression 3> \} ...} \end{array} \right]$$

Therefore, it is assumed that the priority of ADD is the same as those of operators + and -.

(b) When a new operator COMP is defined, as follows,
<expression 8> : = <expression 7> COMP <expression 7> ,
then the priority of COMP is lower than that of operators shown in Fig. 1.

(c) When a new operator MULT is desired, whose priority is between * and **, it is necessary that the operation set shown in Fig.1 should be modified, as follows.
<expression 3> : = <expression 21> [{ * | / } <expression 21>]...
<expression 21> : = <expression 11> MULT <expression 11>
<expression 11> : = <expression 1> [{ ** | $_1$ } <expression 1>]...

(d) An n-tuple operator is illustrated, using the following example.
<expression i> : = <expression j> SUB <expression j>
DIV <expression j>
Two-tuple operator (SUB, DIV) represents a single operation and is used in the above form.

(2) Statement and Expression Semantic Description
The syntax is given by

$$\left\{ \begin{array}{l} \text{<statement-semantic-description-by-a-procedure>} \\ \text{<expression-semantic-description>} \end{array} \right\}$$
: = 1 <identifier> SEMANTICS-OF- $\left\{ \begin{array}{l} \text{STATEMENT} \\ \text{EXPRESSION} \end{array} \right\}$ (<identifier>$_1$)
, 2 { GLOBAL-SECTION | LOCAL-SECTION }
, 3 <character-string-constant> ENTRY $\left\{ \begin{array}{l} \text{nil} \\ \text{RETURNS (<character-string-constant>}_0\text{)} \end{array} \right\}$
, 3 <minor-structure>$_{let}$ [, 3 <minor-structure>$_{let}$] ...
, 2 PROCEDURE
, 3 PARAMETER
, 4 <minor-structure>$_{lem}$ [, 4 <minor-structure>$_{lem}$] ...
, 3 <minor-structure>$_{lep}$ [, 3 <minor-structure>$_{lep}$] ...
, 2 CALL-OF-PROCEDURE
, 3 <minor-structure>$_{lec}$ [, 3 <minor-structure>$_{lec}$] ...;

$$\text{<minor-structure>} \left\{ \begin{array}{l} \text{let} \\ \text{lem} \\ \text{lep} \end{array} \right\} : = \text{<character-string-constant>}_1 .$$

<minor-structure>$_{lec}$: = <parameter>$_1$

where the square bracket $\left\{ \begin{array}{l} \alpha \\ \beta \end{array} \right\}$ specifies that α and β correspond to the specifications on the statement semantic description by a procedure and the expression semantic description, respectively.
The <identifier> is the name which identifies the semantic description of a statement or expression.
The <identifier>$_1$ is used to identify the corresponding statement or expression syntax rule. When the semantics, which have already been defined, are used, only the element with level one should be specified.

The <parameter>$_1$ is information passed from the <parameter>$_1$ of <statement-syntax-rule-description> or <parameter> of <expression-syntax-rule-description> and has the same attributes or forms as those of the <statement-syntax-rule-description> or <expression-syntax-rule-description>.
The parameter passing rule, applicable between syntax definition and semantic definition, is described in Section 4.3.
In this paper, a procedure and function procedure are briefly described only as procedures, in case that it is not necessary to distinguish between them.
The declarations of the procedure and function procedure, which constitute the semantics of statement and expression, respectively, are given in the { GLOBAL-SECTION | LOCAL-SECTION } as follows:
The <character-string-constant> specifies a procedure name.
The <character-string-constant>$_0$ specifies the attribute of return value of the function procedure.
Each <minor-structure>$_{let}$ describes the attributes of a single parameter of the procedure.

(a) When the GLOBAL-SECTION phrase is specified, the <character-string-constant> ENTRY (<parameter-attribute-list>)
$\left\{ \begin{array}{l} \text{nil} \\ \text{RETURNS (<character-string-constant>}_0\text{)} \end{array} \right\}$;
is generated in the global section of the procedure, where the statement or expression is described, and in the global section of the external procedure which contains it. The <parameter-attribute-list> is generated from the list of <minor-structure>$_{let}$.

(b) When the LOCAL-SECTION phrase is specified, the <character string-constant> ENTRY (...) $\left\{ \begin{array}{l} \text{nil} \\ \text{RETURNS (...)} \end{array} \right\}$;
is generated in the local section of the procedure, where the statement or expression is described.
The description of the procedure is generated by the PROCEDURE phrase. The minor structure PARAMETER specifies the parameter list of the procedure. The list of <minor-structure>$_{lep}$ specifies the procedure body.
The procedure name and the attributes of return value of the procedure are given by the { GLOBAL-SECTION | LOCAL-SECTION } phrase.
For the PROCEDURE phrase, the external procedure, beginning from the <character-string-constant> : PROCEDURE (<parameter-list>) $\left\{ \begin{array}{l} \text{nil} \\ \text{RETURNS} \end{array} \right.$
(<character-string-constant>$_0$) $\left. \right\}$;
is generated, where the <parameter-list> is generated from the minor structure PARAMETER.
The procedure reference is generated by the CALL-OF-PROCEDURE phrase. Each <minor-structure>$_{lec}$ specifies a single argument of the procedure argument list.
The procedure name is given by the { GLOBAL-SECTION | LOCAL-SECTION } phrase.
For the CALL-OF-PROCEDURE phrase, the function reference <character-string-constant> (<argument-list>) is generated on

the context where the expression is used.

The statement CALL <character-string-constant> (<argument list>); is generated on the context where the statement is used.

The generated semantics, such as the declarations of a procedure name, the procedure and the reference of the procedure described above, should be allowable in SPL specifications or extended syntax.

(3) Example of an Expression Extension

This example is given in Fig. 4.

When the expression A ADD B occurs in the source program, it is assumed that $1 = A and $2 = B.

In the GLOBAL-SECTION of the procedure, where the expression is described, and of the external procedure, which contains it, the declaration ADDF ENTRY (BIN FIXED, BIN FIXED) RETURNS (BIN FIXED); is generated.

The function procedure shown in Fig. 5 is generated as an external procedure. This shows one of the favorable capabilities of EPL.

The expression A ADD B is replaced by the function reference ADDF (A, B)

This example illustrates that the expression (A*B)/(A+B) is expressed compactly and conveniently by the expression A ADD B.

4.3 Parameter Passing between Syntax Definition and Semantic Definition

Parameter used in the syntax definition is passed to the corresponding parameter of semantic definition.

parameter is denoted as numetir character(s) prefixed by a $ sign. The parameter $n of syntax definition corresponds to the parameter $n of semantic definition.

Parameter relationship between syntax definition and semantic definition is illustrated, using Fig. 3.

In Fig. 3, the parameter $1 of syntax definition FOR-ST is passed to the corresponding $1 of semantic definition SEM-OF-FOR.

In the same way, $2, $3, $4, $5, $6, $7 and $8 of FOR-ST are passed to $2, $3, $4, $5, $6, $7 and $8 of SEM-OF-FOR, respectively.

5. CONCLUSION

In this paper, the hierarchical language system HLS has been discussed with respect to HLS concepts, and language specifications of HLS compo-components. We are planning to use HLS as a tool of software development. In this development, HLS effectiveness should be verified quantitatively.

REFERENCES

(1) Barbara H. Liskov, SPIL: A Language for Construction of Reliable System Software, SIGPLAN notices 8, 9 (September 1973) 100-103.

(2) B.H. Liskov, A Design Methodology for Reliable Software System, AFIPS 1972 FJCC, 41, Part 1, Spartan Books, New York, 191-199.

(3) E.W. Dijkstra, The Structure of the "THE" Multiprogramming System, CACM 11, 5 (1968) 341-346.

(4) E.W. Dijkstra, Go To Statement Considered Harmful CACM 11, 3 (1968) 147-148.

(5) W.A. Wulf, et al., BLISS: A Language for Systems Programming, CACM 14, 12 (1971) 780-790.

(6) N. Wirth, The Programming Language PASCAL, Acta Informatica, 1 (1971) 35-63.

(7) Report of Session on Structured Programming, SIGPLAN notices 8, 9 (September 1973) 5-10.

(8) Victor R. Basili & Albert J. Turner, SIMPL-T A Structured Programming Language, Computer Science Center, Univ. of Maryland (1974).

(9) J.E. Sammet, A Brief Survey of Language Used in System Implementation, SIGPLAN notices 6, 9 (1971) 1-19.

(10) R. Daniel Bergeon, Language for System Development, SIGPLAN notice 6, 9 (1971) 50-72.

(11) B.M. Leavenworth, Programming with(out) the Go To, SIGPLAN notices 7, 11 (1972) 54-58.

(12) E.W. Dijkstra, Notes on Structured Programming, Structured Programming, Academic Press, London and New York (1972) 1-72.

(13) J.D. Gannon & J.J. Horning, Language Design for Programming Reliability, IEEE Transactions on Software Engineering SE-1, 2 (1975) 179-191.

(14) William F. Ross, Structured Programming, Computer 8, 6 (1975) 20-22.

(15) John Naughton, et al., Structured Programming: Concepts and Definitions, Computer 8, 6 (1975) 23-37.

(16) N. Wirth, On the Composition of Well Structured Programs, ACM Computing Surveys 6, 4 (1974) 248-259.

(17) N. Wirth, Program Development by Stepwise Refinement, CACM 14, 4 (April 1971) 221-227.

(18) W.A. Wulf, Programming without the Co To, Proc. IFIP Congress, Vol. 1, North Holland Publ. Co., Amsterdam, The Netherlands, 1972, 408-413.

(19) Jeseph E. Sullivan, Extending PL/I for Structured Programming, Computer Languages 1 (1975) 29-43.

(20) S.L. Stewart, STAPLE, An Experimental Structured Programming Language, Computer Languages, 1 (1975) 61-71.

(21) Proceedings of the International Symposium on Extensible Languages, SIGPLAN notices 6, 12 (December, 1971).

(22) N. Solntseff & A. Yezerski, A Survey of Extensible Programming Languages, Annual Review in Automatic Programming 7 (1974) 267-307.

(23) Bernard A. Galler, Extensible Languages, Information Processing 74, Software (1974) 313-316.

(24) R.H. Leavenworth, Syntax Macros and Extended Translation, CACM 9 (November 1966) 790-793.

(25) IBM System/360 Operating System: PL/I Language Specifications, C28-6571-4, IBM Corp. (1966).

(26) G.M. Weinberg, et al., IF-THEN-ELSE Considered Harmful, SIGPLAN Notices 10, 8 (August 1975) 34-44.

```
< expression 7 > : = < expression 6 > [!(or) < expression 6 >] ...
< expression 6 > : = < expression 5 > [&(and) < expression 5 > ] ...
< expression 5 > : = < expression 4 > [ { < = | < | ¬< | ¬ = | = | > = | > | ¬ > } 
                    < expression 4 > ] ...
< expression 4 > : = < expression 3 > [ { + | - } < expression 3 > ] ...
< expression 3 > : = < expression 2 > [ { * | / } < expression 2 > ] ...
< expression 2 > : = < expression 1 > [ {** | ¬ } < expression 1 > ] ...
< expression 1 > : = < scalar variable > | <constant> | (< expression 7 >)
                    | <function reference >
```

Fig. 1 Scalar expression syntax

```
1.  FOR-ST SYNTAX-OF-STATEMENT,
    2 'FOR',
    2 $1 VARIABLE,
    2 '=',
    2 ITERATION ($2),
     3 $3 EXPRESSION 7,
     3 DISORDER,
      4 OPTIONAL ($4),
       5 'TO',
       5 $5 EXPRESSION 7,
      4 OPTIONAL ($6),
       5 'BY',
       5 $7 EXPRESSION 7,
    2 $8 STATEMENT;

1.  SEM-OF-FOR SEMANTICS-OF-STATEMENT (FOR-ST),
    2 ITERATION ($2),
     3 'DO',
     3 $1,
     3 '=',
     3 $4,
     3 OPTIONAL ($4),
      4 'TO',
      4 $5,
     3 OPTIONAL ($6),
      4 'BY',
      4 $7,
     3 ';',
     3 $8,
     3 'ENDDO;'
```

Fig. 3 Syntax and semantic description example

```
REMOVE COMMENTS:  PROCEDURE;
/* THIS PROGRAM REPLACES ALL SUBSTRINGS BETWEEN '/*' AND
'*/' BY BLANKS */
DATA-SEGMENT;
LOCAL-SECTION;
INPUT CHAR(80); INFILE FILE INPUT; OUTFILE FILE OUTPUT;
(PTR1, PTR2) BIN FIXED;
END-SECT;
END-SEGMENT;
REPEAT FOREVER;
READ FILE (INFILE) INTO (INPUT) ON (EOF) EXIT;
PTR1 = 1;
WHILE PTR1 ¬ = 0
DO /* REMOVE SUBSTRING */;
PTR1 = INDEX (INPUT, '/*');
IF PTR1 ¬ = 0
  THEN /* FOUND BEGINNING */
  PTR2 = INDEX (INPUT, '*/');
  IF PTR2 > PTR1
  THEN  /* FOUND END (AFTER BEGINNING) */
  SUBSTR (INPUT, PTR1, PTR2-PTR1+2) = '  ';
  ENDIF;
ENDIF;
WRITE FILE (OFILE) FROM (INPUT);
ENDDO;
ENDREP;
END REMOVE COMMENTS;
```

Fig. 2 SPL Program Example

```
1  EXP SYNTAX-OF-EXPRESSION 4,
   2 $1 EXPRESSION 3,
   2 'ADD',
   2 $2 EXPRESSION 3;

1  S-OF-EXP SEMANTICS-OF-EXPRESSION (EXP),
   2 GLOBAL-SECTION,
    3 'ADDF' ENTRY RETURNS ('BIN FIXED'),
    3 'BIN FIXED',
    3 'BIN FIXED',
   2 PROCEDURE,
    3 PARAMETER,
     4 'P',
     4 'Q',
    3 'DATA SEGMENT;
      LOCAL SECTION;
      (P, Q, IX) BIN FIXED;
      END-SECT;
      END-SEGMENT;
      IX =
      (P*Q)/(P+Q); RETURN (IX); END ADDF;',

   2  CALL-OF-PROCEDURE,
     3 $1,
     3 $2;
```

Fig. 4 Expression definition example

```
ADDF :  PROCEDURE (P, Q) RETURNS (BIN FIXED);
        DATA-SEGMENT;
        LOCAL-SECTION;
           (P, Q, IX) BIN FIXED;
           END-SECT;
           END-SEGMENT;
        IX = (P*Q) / (P+Q);
           RETURN (IX);
           END ADDF;
```

Fig. 5 Function procedure generated