数理解析研究所講究録
第436巻 1981年 156-173

156

A Language with Modified Block Structure for Data Abstraction and Stepwise Refinement

TAKESHI CHUSHO KENROKU NOGI
Systems Development Laboratory,
Hitachi, Ltd.

TOSHIHIRO HAYASHI
Omika Works,
Hitachi, Ltd.

ABSTRACT----------

A new language was developed in order to support such recent new programming methodologies as data abstraction and top-down development by stepwise refinement without special abstraction mechanisms. The basic structure of this language is introduced by decomposing conventional block structure. That is, shared data are separated from procedures and arranged in a hierarchy of environment modules. A family of procedures is comprised in a process module which is positioned under a suitable environment module.

In addition, type definitions play an important role, not only in data abstraction but also in stepwise refinement of data structure. Depending on these usages, type equivalence is checked by the combination of name equivalence and structural equivalence.

New compiling techniques were required for this language. Separate compilation with complete type-checking is performed by using a library which reserves intermodule information. Compiling speed strongly depends on the data structure of this library which is frequently referred to because of the hierarchy of modules. Object efficiency is also considered, and inline substitution and a compile time facility are implemented.

----

1. INTRODUCTION

For the past few years, considerable effort has been spent on improving software productivity, reliability, and maintainability. This problem has now become more important because of recent rapid increase in the amount of software produced. Research has been dealing with almost every part of the software development process and, above all, the greatest attention has been focused on "structured programming."

The meaning of structured programming has not been clearly defined, but the essence of this approach is to produce a program with high readability. There are well-known techniques for structured programming, such as structured coding[5] and program development by stepwise refinement[6,21]. Abstraction techniques[12] and modularization techniques[15,17] have also been investigated.

For practical use of these methodologies, a programming language supporting them is necessary, and a few new languages have been proposed such as CLU[14], Alphard[22], Mesa[7], Iota[16], Euclid[11], and Ada[9]. The first four are influenced by Simula67 and the remainder by Pascal.

The authors have also developed a new language, SPL[8], for industrial application software with a structure different from the aforementioned languages. This language was designed to support the following techniques:

(1) top-down development by stepwise refinement;
(2) data abstraction;
(3) structured coding.

The first item is a key to a well-structured program. The refinement of a program should be performed by refinement of both data and procedures. Furthermore, at each step of program refinement, the corresponding program should be written. This is because, if a program is written only at the final step, the readability of this program must be lost.

The second item is necessary for reliable data manipulation. A few remarkable abstraction mechanisms have been already proposed, such as **cluster** in CLU, **form** in Alphard, and **package** in Ada. A simple mechanism, however, is preferable because it is not easy for average programmers to master how to use such abstraction mechanisms.

These items (1) and (2) strongly affect the basic structure of a programming language. Although block structure is commonly used as a basic structure, it is not suitable for stepwise refinement and data abstraction because its scope rule is very strong and variable names and procedure names must obey the same

scope rule, as pointed out in paper[19] also. On the other hand, if a totally new structure is introduced, current programmers may hesitate to use the new language.

Therefore, SPL selects a third approach. That is, the conventional block structure is decomposed so that only data obey the static scope rule and procedures do not. Thus SPL was designed while attaching importance to data and has the following features with respect to data:

(1) separation of shared data from procedures;
(2) hierarchy of these shared data for the static scope rule;
(3) data type definition;
(4) strong type-checking.

In relation to these features, new compiling techniques were required. The SPL compiler achieves separate compilation with complete type-checking by using a library which reserves intermodule information. Compiling speed strongly depends on the data structure of this library, which is frequently referred to because of the hierarchical structure of modules.

Object efficiency which may be degraded by new methodologies is also considered, and inline substitution of a procedure and compile time facility are implemented[2]. Checking equivalence of user-defined types is another problem, and it is resolved by combination of name equivalence and structural equivalence.

SPL has already been applied to many industrial systems and its effectiveness in improving productivity and reliability has been confirmed. This paper describes the main features of SPL, new compiling techniques, and its performance evaluations.

2. PROGRAMMING LANGUAGE

2.1 Requirements

In our industrial application software, the conventional high level language has been Fortran which was extended for description of real-time operations. A new language, however, was required for improving productivity, reliability, and maintainability because Fortran-like languages are not suitable

for recent new programming methodologies. The following items are major requirements for the new language:

(R1) Access right to shared data among tasks should be represented more precisely because errors of shared data manipulations often occur.

(R2) The shared data should be sometimes encapsulated by data abstraction mechanisms.

(R3) Top-down development by stepwise refinement of both data and procedures should be supported, as each step corresponds to the individual program.

(R4) Strong type-checking is necessary for high reliability.

(R5) Control statements should be restricted for high readability of control flow.

(R6) A drastic change of programming language is not preferable for avarage programmers who are familiar with conventional languages.

2.2 Decomposition of block structure

Block-structured languages may satisfy a part of the requirements. However, the main defect of block-structured languages is that, by definition, procedures and data obey the same scope rule. For example, consider encapsulating some data and defining several procedures for data manipulation. These procedure names must be available outside a block in which the data are defined, because these procedures should be referred to by users of the data and the data should not. Furthermore, the data must be defined outside these procedure definitions because the data are referred to by these procedures. This is not possible with block-structured languages.

In order to solve this problem, the block structure is decomposed in the new language, SPL, while preserving only a scope rule of shared data. SPL was designed on the basis of this basic structure and has the following features which satisfy aforementioned requirements:

(1) separation of shared data from procedures;

(2) hierarchy of these shared data for precisely representing the scope;

(3) type definition for data abstraction and stepwise refinement of data structure;

(4) global scope of user-defined type names and procedure names;
(5) interface specifications of user-defined types and external procedures which are used without their definitions;
(6) an **exit** statement instead of a **goto** statement;
(7) inline substitution of procedures and compile time facility.

A sample program is given in Figure 1 and Figure 2 for explanation. Figure 1 shows the hierarchy of modules, and Figure 2 shows the definitions of Figure 1 modules P1, E2, and P2. In SPL, there are two types of modules, namely, an environment module and a process module. The environment module is composed of declarations of variables, constants and data types, as well as specifications, from which a tree structure can be constructed to precisely represent these scopes, as shown in Figure 1.

The process module is a set of procedures called functions and is positioned under a suitable environment module as its descendant. Every module can refer to only shared data declared in its ascendant environment modules.

In Figure 2, the new data type STACK and its operation PUSH TO are introduced in P1, and they are defined in the descendant modules E2 and P2 respectively. These are not only a sample of stepwise refinement for data and procedure, but also a sample of data abstraction because the detail structure of the type STACK can be referred to only from P2 including operations for the type.

Item (5) is necessary for complete type-checking. The interface of the procedure ERROR is explicitly specified in E2 because this procedure is referred to without the definition. The control statements in SPL are **if**, **repeat**, **begin/end**, **exit**, **return**, and **stop** statements, and exclude a **goto** statement. The **exit** statement is used for an exceptional exit from a block. Item (7) contributes to the improvement of object efficiency of a modularized program. For example, since the option of the procedure PUSH TO is **open** in P2, the reference

to this procedure in P1 is replaced by its body. At this time, the **%if** statement is executed and object codes for error checking of stack overflow may be emitted.

2.3 Data abstraction

The data abstraction mechanism is simple as is clear from the sample of abstract type STACK shown in Figure 2. That is, the detail structure of a user-defined type can be referred to only from the defining module and its descendant modules, although the user-defined type name and the procedure name can be referred to from every module. According to this scope rule, an abstract type is defined as follows:

(1) The detail structure of the type is defined by using type definition in the lowest descendant module among a hierarchy of environment modules. This type is able to have parameters.
(2) All procedures manipulating data of this type are defined together in a process module, which is only one positioned under the environment module defining the type.

Thus data of this type are encapsulated so that the detail structure can be referred to only from procedures manipulating data of this type. Any module outside these two modules can declare a variable of this type and gain access to this variable by using manipulation procedures for this type.

2.4 Stepwise refinement

SPL is provided with the following features for program development by stepwise refinement:

(1) A type definition for stepwise refinement of data structure.
(2) A program module can be written at each step of program refinement.
(3) A procedure name can be composed of several words in order that the procedure reference may express the function of this procedure as precisely as possible.

A sample is given in Figure 2. After the type STACK and its operation PUSH TO are used in P1, they are refined in E2 and P2 respectively.

Another example is selected from our actual programs. Through stepwise refinement, we developed a program for analysis of paging behavior in an SPL library mentioned later, as reported in paper[2]. The final hierarchy of this program is shown in Figure 3(a). First, variables and constants referred to from every module in the whole system were declared in environment module E_A, and top level procedure was defined in process module A, as shown in Figure 3(b). Then, three functions referred to in A were refined. The second function ANALYZE PAGE_SEQUENCE and its environment module are shown in Figure 3(c). The two variables in E_C are referred to from three functions referred to in C. The data type PAGE-SEQUENCE in E_A was refined in E_B as follows:

```
type PAGE_SEQUENCE = array (MAXNO);
```

3. THE COMPILER

3.1 The outline

The translation of an SPL program can be divided into the following four tasks:

(1) syntax and semantic analysis of a source program;
(2) inline substitution of an **open** procedure;
(3) interpretation of compile-time execution statements;
(4) code generation.

There are two basic problems in the design of an SPL compiler. The first is the processing order of (1), (2) and (3). In the macro facility of an assembly language and the compile-time facility of PL/I, substitution and interpretation are performed on a source program before analysis. This method, however, has two defects that an **open** procedure can not be referred to before defining it, and grammatical errors can not be detected until the expanded program is analyzed. In the SPL compiler, substitution and interpretation are performed together after analysis of a source program, because SPL promotes top-down programming such as the development of a referring module prior to a referred to module.

The second problem is how references to variables, constants, user-defined types, and procedures are connected to

their declarations and definitions. This problem is complicated because the declarations and definitions often belong to a module which is different from a module referring to them and because these modules are compiled separately. In the SPL compiler, a library is introduced in order to reserve intermodule information. Compiling speed strongly depends on the data structure of this library, which will be mentioned later, since it is frequently referred to because of the hierarchy of modules.

As a result, the basic construction of the SPL compiler was designed as shown in Figure 4. An environment module is analyzed by using information from the ascendant environment modules in the library, and then, registered in the library. A process module is dealt with in the same way, and furthermore, it is expanded, interpreted, and translated into object codes if the module includes definitions of procedures with the option **main** or **sub**.

3.2 Separate compilation

Separate compilation is necessary in order to apply modular programming to large-scale software. For example, when applying Pascal for practical use, it is often extended to achieve modularization and separate compilation[10,12]. A key to separate compilation with complete type-checking is the aforementioned library, whose necessity has been discussed a little in CLU[13] and Ada[9] also. The SPL library is composed of an environment module library(EML), a process module library(PML), and a user-defined type table.

3.2.1 EML

The most important point of the design of EML is the representation of the tree structure of environment modules. The simplest way to achieve this structure is to compile every environment module independently and to preserve them and their tree structure individually. In this way, however, a lot of time is spent gaining access to the declarations of variables and constants, because they are searched for in turn from the referring module to the ascendant modules. In the SPL compiler, when an environment module is analyzed, information about the

ascendant modules is extracted from the library as an initial environment. New information about the module under analysis is added to this initial environment and is held in the library as if the environment module and the ascendant modules were originally one module. Thus EML can be referred to quickly, and compiling speed is thereby improved.

3.2.2 PML

PML includes an intermediate language program and an interface table of procedures in addition to the same information as EML. The intermediate language program is emitted by compiling a process module. Each entry of the interface table contains a procedure name, the parameter types, and a pointer to the procedure body translated to the intermediate language, and is used for type-checking of procedure references. This table is reserved separately from the other information in PML for the following two reasons:

(1) When a procedure is referred to before the definition, only the interface specification must be reserved.
(2) Any procedure can be referred to from any module without respect to the place in which this procedure is defined.

3.2.3 The user-defined type table

This table reserves interface specifications and definitions of user-defined types and is independent of EML and PML for the same reasons as in the aforementioned procedure interface table.

3.3 Checking of user-defined types

How to check the equivalence of user-defined types is an important problem because it may restrict the usage of user-defined types. For example, is an assignment statement A=B correct when A and B are declared as follows?

```
type T=integer;
var A:T;
var B:integer;
```

Generally, two definitions of type equivalence are considered[20]. One is called "name equivalence," that is, two variables are considered to be of the same type only if they are declared together or if they are declared using the same type identifier. Therefore, A=B is not correct. The other definition

is called "structural equivalence," that is, two variables are considered to be of the same type whenever they have components of the same type structured in the same way. Therefore, A=B is correct. CLU, Alphard, Mesa, and Ada adopt the former, and Algol68 and Euclid use the latter[1].

The data type definition, however, plays an important role not only in data abstraction but also in stepwise refinement of data structure, and the problem remains no matter which definition is adopted. That is, when using a type definition for data abstraction, the name equivalence is suitable outside the definitions of the type and its operations, because the detail structure of the type should be hidden from the user. On the other hand, the structural equivalence is suitable in the definitions of the operations because the operations for the type must be defined by referring to the detail structure of the type.

The general rule for selection of name equivalence or structural equivalence is considered as follows:

(1) structural equivalence if users of the type need to know the detail structure;

(2) name equivalence if users do not need to know it.

For example, with respect to the user-defined type STACK in Figure 2, module P1 belongs to (2) and P2 belongs to (1).

On the other hand, in SPL, the scope of a user-defined type name is global and the detail structure of the type can be referred to only from the defining module itself and its descendant modules. As a result, the following rule for checking type equivalence is derived from the general rule and this scope rule:

(1) structural equivalence in the module defining the type and its descendant modules;

(2) name equivalence in the other modules.

In CLU and Alphard, limiting to name equivalence causes no problem since the operations for a user-defined type are defined in their abstraction mechanisms, namely, **cluster** and **form**, respectively.

3.4 Inline substitution

Many modules which are referred to from only one or a few places in their ascendant modules come into being after developing a program by stepwise refinement or after decomposing a program into modules so that each module implements one function. If these modules are developed as external ones, their linkage overhead degrades execution speed of the object codes.

In SPL, therefore, a function with the option **open** is inline-substituted. This inline substitution, however, is more difficult than that of a macro assembler or a PL/I compiler. This is because the environment of the referrd to **open** procedure must be unified into the environment of the referring procedure, and because the expanded intermediate language program must be connected to this unified environment.

In order to simplify this manipulation, a pointer from an intermediate language program to the environment is composed of two parts, namely, an entry address of a module management table and a location address in each environment module. Therefore, the connection between the unified environment and the expanded program is performed only by rewriting the module management table, and it is not necessary to modify the intermediate language program. Figure 5 shows the mechanism by which the intermediate language of references to the shared data A points to the unified environment when procedure F2 is expanded into procedure F1.

Another problem of inline substitution is how to identify the version of environment modules which are included in both the referring module and the referred to module. In Figure 5, E0 and E1 are included in both process modules P1 and P2. However, it is not guaranteed that the versions of E0 and E1 in P1 are the same as the ones in P2. For this checking, every compiled module is given a unique number corresponding to its birthday.

4. PERFORMANCE ANALYSIS

4.1 Compilation time

The SPL was implemented on the control computer HIDIC 80 as a preprocessor of the real-time Fortran. The compilation time was analyzed with a tested program which was in practical use. Part of the results are shown in Table I. The analysis phase uses 2/3 of the cpu time and the substitution and interpretation phase occupies nearly 10%, although this ratio depends on each program feature.

Another interesting item is access time to auxiliary memory (a magnetic drum), since the SPL library in this device is expected to be frequently referred to because of separate compilation. This access time to the auxiliary memory is equal to 17% of the cpu time, and most of that time is spent at the analysis phase because environment modules are separated from process modules and a hierarchy is constructed. Therefore, paging behavior of the SPL library was analyzed in detail as reported in [4].

4.2 Object efficiency

The program developed by stepwise refinement is composed of many self-contained modules. Therefore, degradation of object efficiency may be caused by the following:

(1) linkage overhead
(2) redundancy
(3) increase of local variables

In SPL, however, these defects can be avoided. That is, the linkage overhead is reduced by inline substitution of **open** procedures[2]. This effect has been reported in CLU[18], too. Redundancy arises because a referring procedure is written without knowledge of the detail process of a referred to procedure. This redundancy can be avoided by using compile time facility. The compiler optimizes the storage allocation of local variables, which can share storage with each other. The result, as confirmed by several experiments, is that degradation of object efficiency in SPL is less than 10% in comparison with real-time Fortran.

5. CONCLUSIONS

The new language SPL and its compiler were developed in order to support such programming methodologies as data abstraction and top-down development by stepwise refinement. The basic structure was introduced by decomposing conventional block structure. That is, the static scope rule of block structure for shared data was retained with the modification that the declarations of shared data are separated from procedures and arranged in a hierarchy. However, procedure names have global scope.

Furthermore, new compiling techniques were required for the SPL compiler with respect to this basic structure and other remarkable features of the language. These techniques were confirmed to be effective by performance evaluation of the compiler.

The first version of the language SPL was designed in 1975 and later updated. The compiler was developed in 1976 and has been further improved. Further study is needed to develop various supporting tools for this language such as an interactive programming system for restructuring and optimization of source programs[2,3] and a manipulation system for the SPL library.

ACKNOWLEDGEMENTS

The authors wish to express their gratitude to Yutaka Takuma for providing the opportunity to conduct this study. They are also indebted to Dr. Ikuo Nakata for his invaluable technical assistance and Kiyomi Mori and Kazuo Katogi who implemented the SPL compiler with the authors.

REFERENCES

1. BERRY,D.M. AND SCHWARTZ,R.L. Type equivalence in strongly typed languages: one more look. SIGPLAN Notices 14,9(Sep. 1979), 35-41.
2. CHUSHO,T.,AND HAYASHI,T. Two-stage programming: interactive optimization after structured programming. In Proc. 3rd

USA-Japan Computer Conf., Oct. 1978, pp.171-175.

3. CHUSHO,T. A good program = a structured program + optimization commands. In Proc. IFIP80, Oct. 1980, pp.269-274.
4. CHUSHO,T.,AND HAYASHI,T. Performance analysis of paging algorithms for compilation of a highly modularized program. IEEE Trans. Softw. Eng. SE-7,2 (March 1981), 248-254.
5. DIJKSTRA,E.W. Goto statement considered harmful. Comm. ACM 11,3 (March 1968),147-148.
6. DIJKSTRA,E.W. Note on structured programming: Structured Programming. Academic Press,London and New York,1972, pp.83-174.
7. GESCHKE,C.M.,MORRIS,J.H.Jr AND SATTERTHWAITE,E.H. Early experience with Mesa. Comm. ACM 20,8 (Aug. 1977),540-553.
8. HAYASHI,T.,et al. Top-down structured programming language for real-time computer systems - SPL. Hitachi Review 26,10 (Oct. 1977), 333-338.
9. ICHBIAH,J.D.,et al. Rationale for the design of the Ada programming language. SIGPLAN Notices 14,6 (June 1979).
10. KIEBURTZ,R.B.,BARABASH,W. AND HILL,C.R. A type-checking program linkage system for Pascal. In Proc. 3rd Int. Conf. on Softw. Eng., May 1978, 23-28.
11. LAMPSON,B.W. et al. Report on the programming language Euclid. SIGPLAN Notices 12,2(Feb. 1977).
12. LEBLANC,R.J. AND FISCHER,C.N. On implementing separate compilation in block-structured languages. In Proc. ACM Symp. Compiler Construction, Aug. 1979, 139-143.
13. LISKOV,B.,SNYDER,A.,ATKINSON,R. AND SCHAFFERT,C. Abstraction mechanism in CLU. Comm. ACM 20,8 (Aug. 1977), 564-576.
14. LISKOV,B. et al. CLU Reference Manual. MIT/LCS/TR-225, M.I.T., Cambridge, Mass. , Oct. 1979.
15. MYERS,G.J. Reliable Software Through Composite Design. Petrocelli/Charter, New York, 1975.
16. NAKAJIMA,R.,HONDA,M. and NAKAHARA,H. Hierarchical program specification and verification - a many-sorted logical approach. Acta Informatica 14(1980),135-155.
17. PARNAS,D.L. On the criteria to be used in decomposing systems into modules. Comm. ACM 15,12 (Dec. 1972), 1053-1058.
18. SCHEIFLER,R.W. An analysis of inline substitution for a structured programming language. Comm. ACM 20,9 (Sept. 1977), 647-654.

19.TOWSTER,E. A convention for explicit declaration of environments and top-down refinement of data. IEEE Trans. Softw. Eng. SE-5,4(July 1979), 374-386.

20.WELSH,J., SNEERINGER,W.J. AND HOARE,C.A.R. Ambiguities and insecurities in Pascal. Software-Practice and Experience, 7, 6 (Nov.-Dec. 1977),685-696.

21.WIRTH,N. Program development by stepwise refinement. Comm. ACM 14,4 (April 1971), 221-227.

22.WULF,W.A.,LONDON,R.L. AND SHAW,M. An introduction to the construction and verification of Alphard programs. IEEE Trans. Softw. Eng. SE-2,4 (Dec. 1976), 253-265.

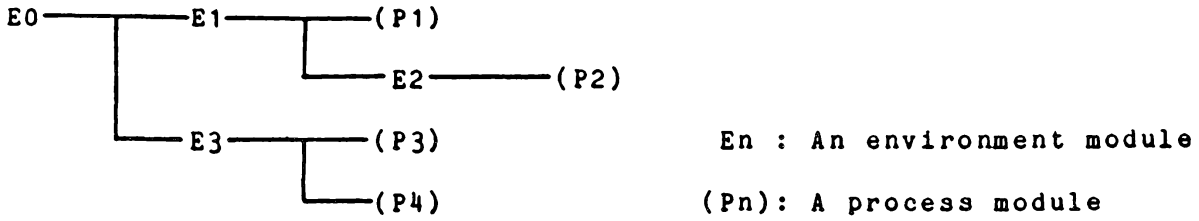


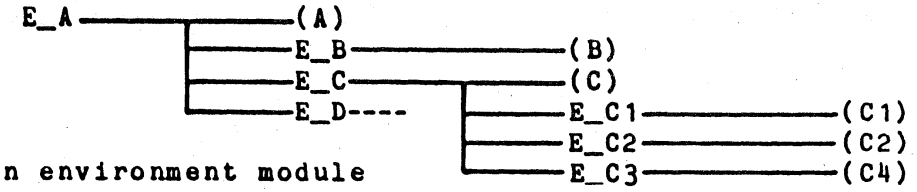
Fig. 1. A sample of hierarchical structure of an SPL program.

```
process P1(E1);
 func EVAL opt(sub);
  var S1:STACK(100);
  .....
   PUSH(VALUE) TO(S1);
  .....
end P1;

environment E2(E1);
 specification;
  func ERROR(CODE:int) opt(sub);
 end;
 declaration;
  type STACK(LENGTH:int)
            =(TOP:int init(0),
              STK(LENGTH):real,
              MAX:int init(LENGTH));
 end;
end E2;
```

```
process P2(E2);
 func PUSH(V) TO(S) opt(open);
  par V:real,
      S:STACK(*);
   S.TOP=S.TOP+1;
   %if LEVEL
     is 1 then
          if S.TOP > S.MAX
            then ERROR(E013);
            else S.STK(S.TOP)=V;
          end;
     is 2 then S.STK(S.TOP)=V;
    end;
 end PUSH;
 .....
end P2;
```

Fig. 2. Samples of SPL modules.



E_aa : An environment module
(aa) : A process module

(a) A hierarchy of modules.

```
environment E_A;
 dcl;
  var   PSIZE:int,
        NPAGE:int;
  var   PSEQ :PAGE_SEQUENCE,
        NDATA:int;
  const MAXADR = 16384,
        SSIZE  =    64,
        MAXPG  =   256;
  const ENDMK  =    -1;
 end;
end E_A;
```

(b) Top level modules.

```
process A(E_A);
 function PAGING opt(main);
  for PSIZE=SSIZE, SSIZE*8 through PSIZE*2
    repeat
       INITIALIZE PAGING;
       ANALYZE PAGE_SEQUENCE;
       APPLY PAGING ALGORITHMS TO PAGE_SEQUENCE;
    end;
 end PAGING;
end A;
```

```
environment E_C(E_A);
 dcl;
  var LTABLE(MAXPG):int,
      STABLE(MAXPG):(FREQ:int,
                     PGNO:int);
 end;
end E_C;
```

(c) Part of second level modules.

```
process C(E_C);
 function ANALYZE PAGE_SEQUENCE opt(open);
    EXAMINE LRU_PROPERTY;
    EXAMINE STATIC REFERENCE FREQUENCY;
    COMPARE LRU WITH STATIC;
 end ANALYZE;
end C;
```

Fig. 3. An example for program development by stepwise refinement.

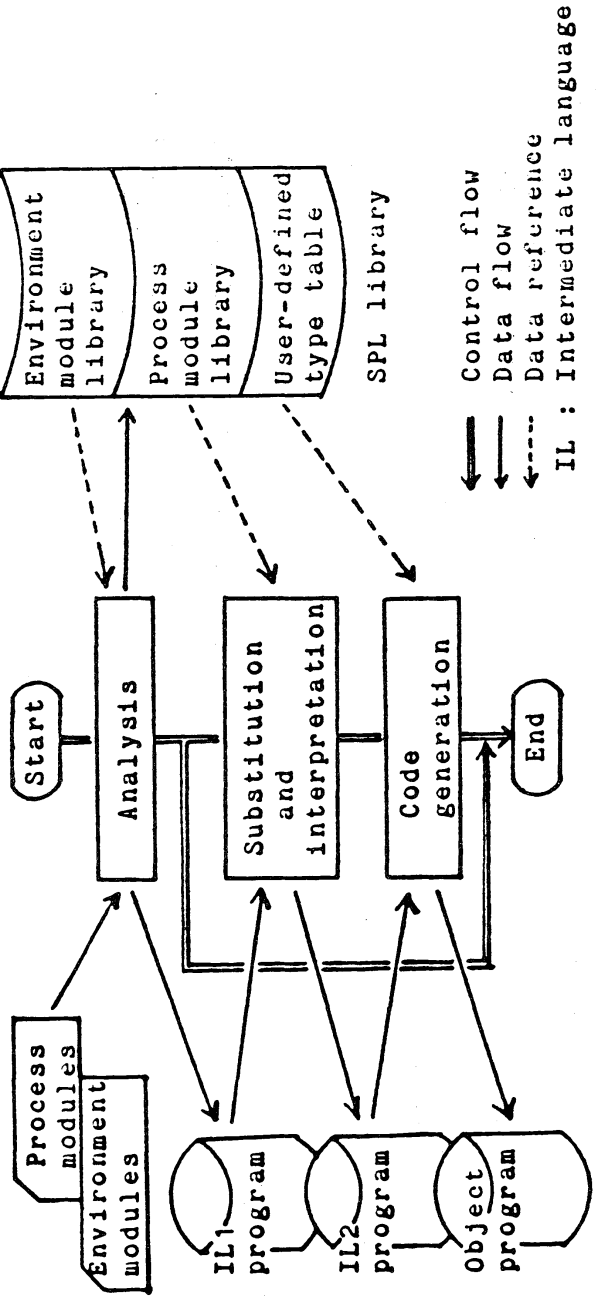
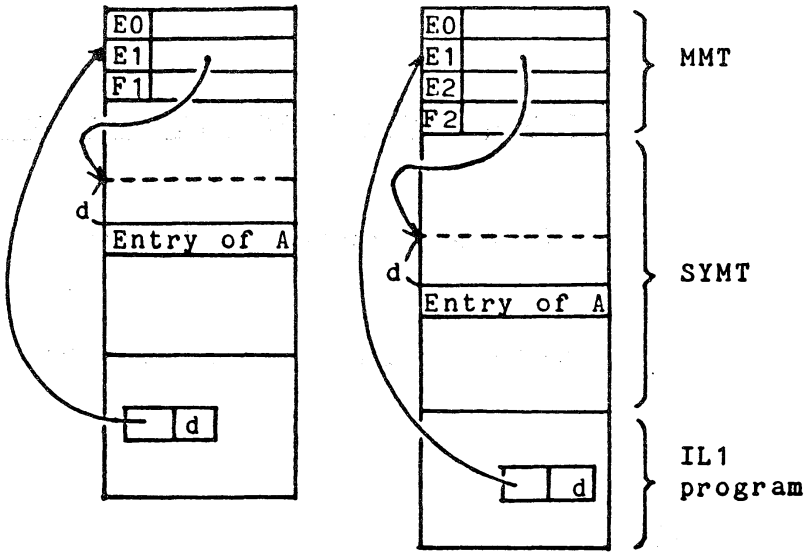


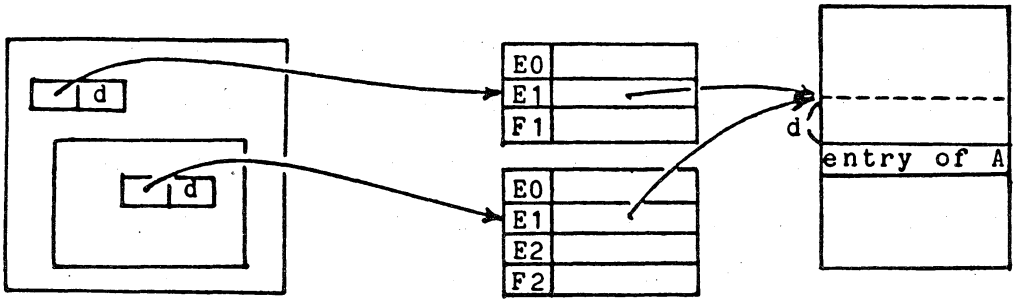
Fig. 4. Process flow of the SPL compiler.

```
environment E0;
 .....
environment E1(E0);
 .....
  var A;
 .....
process P1(E1);
  func F1 opt(sub);
    ...A...
      F2;
    .....
environment E2(E1);
 .....
process P2(E2);
  func F2 opt(open);
    ...A...
 .....
```

(a) A sample program.



(b) P1 and P2 modules in PML



IL1 program MMT SYMT

(c) The internal structure of the expanded program.

MMT : Module management table
SYMT : Symbol table

Fig. 5. A mechanism for inline substitution.

Table I. Analysis of Compilation Time for a Sample Program

Phase	CPU time(sec)	M/D access time(sec)
Analysis	69.42 (67.5%)	14.92 (85.2%)
Substitution & interpretation	8.88 (8.6%)	0.97 (5.5%)
Code generation	24.62 (23.9%)	1.63 (9.3%)
Total	102.92	17.52