A SOFTWARE DESIGN SYSTEM

Thesis by

Gideon David Hess

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1980

(Submitted March 13, 1980)

ACKNOWLEDGMENT

I wish to thank Dr. Frederick B. Thompson for his patient guidance and help during my graduate studies at Caltech.

ABSTRACT

The goal of the research described in this thesis was to build a system that supports, without interfering with, the activity of systematic software design and takes upon itself mechanical activities the designer can be spared.

Two of the main activities which constitute the process of software creation are:

1. Designing a solution to the problem.

2. Implementing the design.

The activity of design has to be performed by the programmer himself, it can only be aided by the computer. Producing a program from a complete design is a mechanical activity the computer can take upon itself.

These observations lead to the following objectives that a software design system should meet:

1. Providing tools that support the design activity and enable maximum flexibility.

2. Recognizing the lowest level primitives of the design as the target language and producing the program in this language.

A system along these guidelines was implemented. It permits the user to write definitions which refine high level design decisions into lower levels and, at the same time, serve as syntax descriptions and translation rules for the languages used in the design.

The system operates in two user-controlled passes. In the first pass the user's definitions are read, either interactively or from external files, and the syntax rules are stored in a dictionary. In the second pass a syntax driven language processor uses the dictionary to compile the user's program into the target language which consists of the lowest level constructs of the design.

Due to the freedom the programmer has in design, several kinds of syntactic ambiguities may be introduced with - or without - the user's attention. Unless caused by user errors, the translator tries to resolve these ambiguities to match the designers intentions.

In order to reduce the amount of time and space required for parsing, long texts are divided into subtexts which are translated separately. Guidance as to which subtexts are separately translatable is provided by the user in a natural way by composing the design of statements.

A command language enables the user to control the passes, to look at the contents of the dictionary and of external files, to monitor the translation process for debugging purposes, to store dictionaries for later use and retrieve them and to modify special symbols used in definitions.

The system is implemented in Simula. A second system is presently being implemented as part of POL (Problem Oriented Language), a system for writing and using application languages. POL's metalanguage enables the user to build - or extend - object languages by writing new syntax rules. The tools of the development system described above are incorporated into the metalanguage in order to aid the application programmer in the design and compilation of the semantic routines of these rules.

Table of Contents

1 INTRODUCTION

The goal of the research described in this thesis was to build a system that supports, without interfering with, the activity of systematic software design and takes upon itself mechanical activities the designer can be spared.

Both in software and in other engineering disciplines designers have to use divide-and-conquer strategies. One can grasp only a limited amount of depth of a system's complexity at one time and it is this limitation that determines the nature of design activity. If, for example, you look at an airplane from a distance that allows you to see the whole of it, you may see general features like the general shape of the wings, to which part of the fuselage they are attached, and how many engines the airplane has. But details like the exact curvature of the wings, the number of bolts attaching them and the size of these bolts can not be seen from this point of observation. The designer of the airplane faces the same problem. The depth of his grasping ability is limited and he is forced to abstract details while doing the overhaul design and to forget about the overhaul when dealing with details and, since both are interdependent, iterate between them.

Software designers face the same kind of quantitative problems, which call for the same kind of approach to the design activity, but there seem to be difficulties in implementing this approach. One only has to look at the large amount of literature about the subject to realize that it is not as natural and straightforward as the comparison with other areas might suggest.

Two reasons for this difference are pointed out by Bauer in [3]. The first one is the abstract nature of software products as opposed to those of other engineering disciplines:

> Software is not a physical object, it is non-material. ... Software is an abstract web, comparable to mathematical tissue, but it is a process and in so far very different from most of usual mathematics, too.

The second reason is that software lacks a sound foundation of research and development. A foundation that most other engineering disciplines have and utilize:

> A hasty buildup in the computer industry has not provided the best climate for satisfactory development of good software. ... We need a more substantial basis to be taught and monitored in practice on the structure of programs and the flow of their execution...

The term "software development" covers a broad spectrum of activities like: Specifications writing, design of a solution, coding, compilation, debugging, verification and documentation. The work described in this writeup is an attempt to take a closer look at the design and coding stages of software development, find the problems involved and provide solutions to them. Chapters 2, 3 and 4 analyze these activities, discuss other systems in this area and come up with a series of objectives for a software design system. Chapters 5 through 10 describe the system and issues regerding its development, operation and use.

Chapter 11 summarizes those issues in short. A user's manual, details about the system's implementation and an example of its use are given in the appendices.

2 ORGANIZATIONS IN SOFTWARE

2.1 ORGANIZATION OF DESIGN

Designing a software system is the action of bridging a gap between the system's specifications on the one hand and a machine on which it has to run on the other hand.

By "machine" I refer to an entity that represents all the computer-side factors the designer has to take into account. This includes the programming language to be used as well as all the performance characteristics relevant to the particular task such as speeds and capacities.

Ideally the specifications are independent of this machine. They merely represent the requirements from the system so that its performance meets the user's expectations. In reality, since machines are not ideal, requirements may be impossible - or very difficult - to achieve. As a result compromises are often necessary and even then the software product may have to undergo various cycles of performance checks and modifications.

The size of the gap depends mainly on the size of the system, but also, to a large extent, on the machine. A large gap can not be bridged all at once, it has to be done step by step. The number of ways by which this can be achieved is large and, as was pointed out in the introduction, it is not always clear what the best way to divide the task is. However there are objectives the design should meet.

The design should be as simple and manageable as possible. In order to achieve this the number of simultaneous decisions that have to be made at each step should be minimized.

Another objective, mentioned by Goos in [5] is that the design should proceed in such a way that one should be able to convince himself at every stage that what has been done so far is correct, and should not have to revise large earlier designed parts because of errors detected at the present stage.

The design strategy that meets these objectives is the one Dijkstra describes in "Notes on Structured Programming" [10]. A general formulation of the problem, is divided into a small number of sub-problems each of which is further divided until finally the building blocks become visible at the bottom.

One would like this activity to proceed in an orderly top-down manner, this might even be the case if an experienced programmer tackles a small problem, but in reality one can not grasp all aspects of a large system at one time and hence is unable to predict all the implications of design decisions. This gives rise to numerous cases where segments of the design are started at the bottom - or an intermediate - level rather than at the top, parts have to be reviewed, rewritten, modified or generalized because they do not perform the required tasks or do not match all other parts with which they are related. The designer may have to move back and forth, changing, adapting, compromising until all parts work correctly together.

Here are a few examples for the kinds of activities which constitute a design process.

In the report about the REL system [35] the issue of parallel vs serial syntactic and semantic processing in data base query systems is tackled. It had often been suggested that the semantic processing of a sentence should be performed in parallel with its parsing so that its parts parse in two ways, one of which can be eliminated by the semantics, reducing the number of spurious parses in further syntax processing. As it turned out in experiments, since some of the spurious parsings have a semantic meaning, this method resulted in numerous superfluous references to the data base causing disk accesses whose time exceeded the time saved by reducing spurious syntactic analysis which can be performed in main memory. Had the data base been small enough relative to main memory, or had the semantics been of a different nature, the parallel processing scheme might have worked. This is an example of how performance of parts which are low level in the systems hierarchy can influence and overthrow high level decisions.

While programming SDS I often realized that a sequence of statements, I was just about to write, appeared in at least three other places in my system and decided to make a procedure out of it. Often these statements included rather long and delicate expressions and were susceptible to trivial errors and omissions. After writing and carefully checking the procedure I returned to the places where it should have been in first place and replaced the sequence of statements by the safer procedure call.

It often happens that parts of the system which have been designed to perform a particular function are generalized and used for previously unplanned purposes. For example the SDS system includes a procedure that dumps the dictionary contents on the terminal. At a later stage it turned out to be desirable to dump only certain parts of the dictionary in some cases. Rather than writing a new procedure it was easier to generalize the existing one and add another argument to it, changing its calling sequence. This in turn required modifications wherever the procedure was called.

Once a system is completed and running it usually undergoes a process of polishing and optimizing. In polishing one takes care of issues that have been neglected so far because they are not crucial to the system's operation such as nice input and output formats and corrections of errors that could be lived with so far. The process of optimizing consists of performance measurements, attempts to detect bottlenecks and make them more efficient by careful reprogramming, often in assembly language.

This was just a short and far from complete list of examples of the kinds of activities that constitute a software design process and prove that describing it as a tree may conform to our desire and aesthetical tendencies, but, in most cases, not to reality.

2.2 ORGANIZATION OF THE SOFTWARE SYSTEM

The following objectives, mentioned in [12], relate to the final product of the design:

-The system should be organized in such a way that will enable a larger group of people to participate in the design. The amount of necessary communication between members of the group should be kept to a minimum and it should be as clear as possible.

-It should be easy to modify and to maintain the system. Changes in one part should not cause a chain reaction of many other necessary modifications and the consequences of a change on the rest of the system should be easy to predict.

These objectives imply that the system should be divided into parts which are as independent from each other as possible and that the necessary dependencies have to be clearly defined and easily understood.

Organizing the system's modules in a tree structure would certainly meet these requirements, but it turns out that most systems have a more complicated organization which can not be represented by a simple tree.

The system's organization is determined by its designer. If several people received equal specifications and used the same programming language, their organizations would certainly differ, yet all would have two things in common: Being externally prescribed, the specifications on one side, and constructs of the programming language on the other, would form the top - and bottom - levels, respectively, of all organizations. Besides representing both extreme sides of the system's hierarchy, these two levels also represent both extremes of another scale, namely that of specialization. Consisting of the system's overall specifications the top level is unique for a particular system and as such represents the highest level of specialization. The bottom level, on the other extreme, consists of tools which are shared by all parts of the system, in fact - by all systems written in the same language, and thus represents the lowest specialization level. Between the two extremes, moving down the hierarchy one observes a shift from system oriented to programming-tool oriented parts. The lower the part is the more its use tends to be shared by others. Therefore the organization is composed of layers of modules rather than being tree like. Each module occurs only once and it refers to modules of layers below it.

Another factor that affects the organization is the fact that most high level programming languages support recursion. A module may refer to itself or several modules may refer to each other in cycles. Thus in addition to references between layers there are references between parts within layers.

The diagram on page 11 includes a crude description of the organization of SDS. It is shown here as an example for the characteristics of a system's organization. An arrow between boxes indicates that parts within the box from which it emanates refer to parts of the other. The bottom box includes the programming language constructs. All other boxes make use of it, but the corresponding arrows were omitted in order to keep the diagram legible. The second level includes text utilities, parts of which may perhaps be special to this particular system, but within it they are used by most modules. Proceeding up the diagram modules become more amd more specialized and unique to the system.

2.3 RELATIONSHIP BETWEEN THE TWO ORGANIZATIONS

The first two paragraphs of this chapter discussed the organization of a software project from two aspects. One is the conceptual division of the task into sub-tasks by the designer during the design process. The second organization is that of the software product itself. How are these two related? Which of the sub-tasks the designer had in mind really becomes a module of the system? (By "module" I refer to entities like Algol procedures, Fortran subroutines, Simula classes or whatever mechanisms a programming language provides to divide the program).

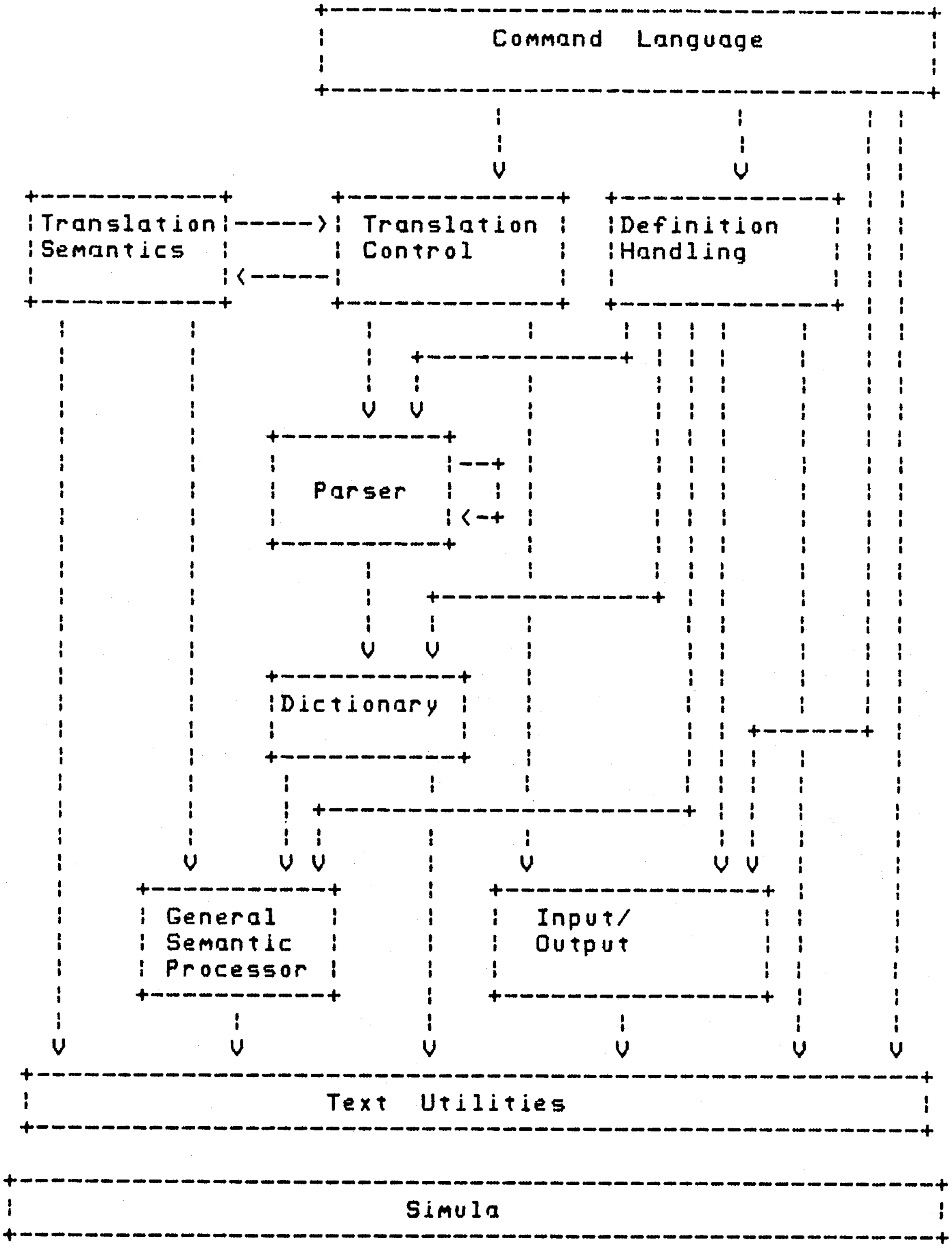

Figure 1: Organization of a software system

To answer this question look first at the two extremes. On the high-level side the system itself is a module. In most cases its high-level sub-tasks are modules (all the boxes in the diagram on page 11 are modules with various depths). On the other side there are the constructs of the programming language. Making every statement into a module (eg an Algol procedure) would be absurd. Not only would it be highly costly due to the large overhead in time and space, it also would not contribute anything to the design process or to a better understanding of the system, in fact - it would make it much more cumbersome. An important part of the design is constituted by the decisions as to which design blocks will have matching system modules and which blocks are only conceptual sub-tasks whose mere purpose is to simplify the design process.

Assembly languages allow this kind of distinction by permitting the user to write macros and procedures. Both are tools that enable the programmer to look at the program from a higher point of view. Once written they can be used as higher level constructs ignoring their details. Whenever a macro is called, the sequence of instructions is substituted into the code, while procedure calls stay in the program as such and the procedure itself exists as a separate entity. The decision as to whether a design block should be a macro or a procedure is based upon time and space considerations. Here is a typical example taken from the REL system which is written in IBM assembly language. The overhead for procedure calls in REL is about 30 instructions. In order to obtain a new list element in register R the following instructions have to be performed:

```
R := top of available-space-list;
If R = nil then
Begin
    Garbage-collect;
    R := top of available-space-list;
End;
Top of available-space-list := Next free list-element;
```

Being very widely used in the system, this sequence is an ideal candidate for a separate design block. If there exists an available list element (which is mostly the case), the process takes three instructions. Writing it as a procedure would increase the number of instructions executed almost each time by an order of magnitude and substantially slow down the whole system, therefore it is defined as a macro. If there are no available list elements, the garbage collector is called. The garbage collector contains a few hundred instructions, compared to which the procedure call overhead is small. On the other hand - substituting its whole text whenever a list element is required would largely increase the program size, therefore it is defined as a procedure rather than as a macro.

Blocks of the design organization which are also blocks of the system organization will be referred to as procedures throughout the rest of this thesis. Blocks of the design organization which have no corresponding blocks in the system will be referred to as macros.

In order to obtain the system organization from the design organization, one has to know which of the design blocks are procedures and which are macros. Every reference to a macro in a design procedure has to be replaced with the actions in the blocks the macro refers to, and this rule is to be applied recursively. Completion of this process for all the macros results in the actual procedures of the software system.

For example consider the diagram on page 15. It is an enlarged section of the diagram on page 11. Every box is divided into two parts: The first part is a header which states whether it is a procedure or a macro and also includes the action it performs. The text describing the action serves as the calling sequence within the design organization. The second part is a refinement of this action. The line originating at the box points to the boxes which correspond to the refinements. The bottom blocks contain constructs of the programming language. They can be considered macros which refine into themselves. The diagram on page 16 shows the corresponding section of the system organization which is obtained by collapsing the macros.

```
                      +---------------+
                      ¦ Procedure     ¦
                      ¦ Getfile       ¦
                      ¦---------------¦
                      ¦ Initialize    ¦
                      ¦ Get file name ¦
                      ¦ Open file     ¦
                      ¦ Terminate     ¦
                      +---------------+
                              ¦
                              ¦
      +--------------------------------------------+
      ¦           ¦                ¦               ¦
      ¦ +------------------+       ¦         +-----------+
      ¦ ¦Macro             ¦       ¦         ¦Macro      ¦
      ¦ ¦Get file name     ¦       ¦         ¦Terminate  ¦
      ¦ ¦------------------¦       ¦         ¦-----------¦
      ¦ ¦Request file name ¦       ¦         ¦Getfile:-F ¦
      ¦ ¦Read file name    ¦       ¦         ¦End;       ¦
      ¦ +------------------+       ¦         +-----------+
      ¦           ¦                ¦
      ¦           ¦      +--------------------+
      ¦           ¦      ¦Macro               ¦
      ¦           ¦      ¦Open file           ¦
      ¦           ¦      ¦--------------------¦
      ¦           ¦      ¦F:-new infile(N);   ¦
      ¦           ¦      ¦F.open(blanks(80)); ¦
      ¦           ¦      +--------------------+
      ¦           ¦
      ¦     +--------------------------------+
      ¦     ¦                                ¦
      ¦ +------------------------+   +----------------+
      ¦ ¦Macro                   ¦   ¦Macro           ¦
      ¦ ¦Request file name       ¦   ¦ Read file name ¦
      ¦ ¦------------------------¦   ¦----------------¦
      ¦ ¦Outtext("File name: "); ¦   ¦N:-Input;       ¦
      ¦ ¦Breakoutimage;          ¦   +----------------+
      ¦ +------------------------+           ¦
      ¦                                      ¦
+--------------------------------+     +----------+
¦Macro                           ¦     ¦Procedure ¦
¦Initialize                      ¦     ¦Input     ¦
¦--------------------------------¦     ¦----------¦
¦Ref(infile) procedure getfile;  ¦     ¦  .       ¦
¦Begin                           ¦     ¦  .       ¦
¦Text N; Ref(infile) F;          ¦     ¦  .       ¦
+--------------------------------+     ¦          ¦
```

Figure 2: An enlarged section of figure 1

```
+-----------------------------------+
! Ref(infile) procedure getfile;    !
! Begin                             !
! Text N; Ref(infile) F;            !
! Outtext("File name: ");           !
! Breakoutimage;                    !
! N:-Input;                         !
! F:-new infile(N);                 !
! F.open(blanks(80));               !
! Getfile:-F;                       !
! End;                              !
+-----------------------------------+
                 !
                 !
+-----------------------------------+
!  ... Input ...                    !
!                                   !
!                                   !
```

Figure 3: Section of the system organization

2.4 HIERARCHIES OF LANGUAGES

This paragraph presents two hierarchies of languages related to the design - and system - organizations discussed so far.

One way to look at the language hierarchy is from the point of view expressed in [25]. A language is seen as a formal means of expressing one's current view of his world. The language is a function of this world. As the person's attention shifts, although the syntactic structure of his language does not necessarily change, the vocabulary may increase or decrease, old words may get new interpretations.

A notion formally introduced in [25] is that of degrees of expressiveness. It corresponds to the intuitive idea about expressiveness, namely that language L1 is at least as expressive as language L2 if L1 can distinguish between all states that can be distinguished between by L2. Thus speaking in L1 one may be able to give a more refined description of events described by L2 and talk about things which are irrelevant for L2's user.

The thought process one undergoes during design activity corresponds, using these notions, to a progression through a series of languages as the designer moves between different points of view of the system. At some points details of a small part, like how to compare two texts, are worked out while all other aspects are ignored. At another moment details that affect the behaviour of other parts, for example - what should the exact format of list elements be, are designed. At yet onother moment some high level decision may be made while exact details of its implementation remain disregarded. The sets of states of the corresponding languages shrink, expand or shift one way or the other, hence the languages themselves form a partially ordered set in regard to degrees of expressiveness. It is like looking at a drawing from different distances. A very close look may show dots and lines of paint, but be too close to take in the entire picture. At a larger distance, though those dots are still visible, the objects they form can also be seen. At a very large distance the dots disappear from the view to give way to an overall impression of the picture.

Since the design organization may be created in any order, there is no way of telling through what languages the designer progressed by merely looking at the complete design. Moreover, since the design activity may be iterative, there could be languages which existed during the design but left no trace in the final organization.

To see the second language hierarchy consider the steps one has to go through in writing the program, given the design graph. The final program consists of the procedures, written in the target language, consisting of the lowest level constructs of the design. In order to arrive at the program one has to traverse the graph starting at the top and for each procedure-node go through a translation process in which every macro-node is replaced with its immediate descendants. In doing so one progresses through a series of languages for each procedure, and, since the constructs of each language describe refinements of those of the higher languages, the sequence is ordered in increasing expressiveness.

The union of all the lowest level languages of the procedures forms the target language of the translation process. This target language is not necessarily identical with the programming language. It contains all the constructs from which the procedure bodies are built, namely programming language constructs and procedure calls.

For example, in the translation of the procedure Getfile from the design organization of figure 2 one goes through four languages. The text:

```
Getfile
```

is in language 1.

The texts:

```
Getfile
Initialize
Get file name
Open file
```

Are in language 2.

The texts:

```
Ref(infile) procedure getfile;
Begin
Text N;
Ref(infile) F;
Request file name
Read file name
F:-new infile(N);
F.open(blanks(80);
Getfile:-F;
End;
```

Are in language 3.

The texts:

```
Ref(infile) procedure getfile;
Begin
Text N;
Ref(infile) F;
Outtext("File name: ");
Breakoutimage;
N:-Input;
F:-new infile(N);
F.open(blanks(80));
Getfile:-F;
End;
```

Are in language 4.

Note that both the depth of this language hierarchy and the depth of the system organization depend, in opposite directions, on the number of macros in the design. When the system organization is derived from the design, all the macros collapse and disappear. At the same time every application of a layer of macros introduces another intermediate translation step and hence - another language. Therefore the greater the number of macros among the design blocks the flatter will the system be and the longer the sequence of languages.

3 OBJECTIVES

The preceding chapter covered issues of software development which give rise to a set of objectives for the construction of a software development system as stated in the following paragraphs.

3.1 STRUCTURED, BUT FLEXIBLE, DESIGN

Design, by its nature, is a creative process where one invents, finds alternatives, tries them and chooses between them, reviews and changes earlier designed parts. A software development system has to support this maze of activities and at the same time has to enable the user to find his way through them. Thus it has to allow maximal flexibility and impose a minimum of structure.

In order to be more precise about it, refer to the design organization discussed in paragraph 2.1. The structure that a system should support is the structure of this organization: actions that are broken down into parts in an open-ended process of stepwise refinement possibly with actions referring to themselves and actions referred to by more then one other action. Within this structure maximal flexibility should be provided. The designer should be able to build the organization from any desired starting point and from each point - to move in any direction: down - by refining already designed steps, or up - using already designed steps in new, more abstract, ones. The system should also allow iterations by enabling the user to cancel previously made decisions or to redo them.

Another aspect of flexibility is the choice of a language. Designers may want to use different languages like natural English, semi-formal English or some formal programming or design language. Some may mix languages and use diffferent ones for different steps of the design or for different moods of the designer. Any kind of language should be permitted and accepted by a system that supports the action of design.

3.2 AUTOMATED CODING

As described in chapter 2, the design organization is, in general, different from the actual program. The transition between the two is quite straightforward: in order to derive the program from the design, one has to decide which step is a macro and which is a procedure and to write the program in the target language accordingly, by applying the macros to the procedures. This process can be automated if the macro - procedure decisions are provided.

A second objective of a software development system is to alleviate the user of the coding task by letting him incorporate the macro - procedure decisions into the design so that the system can produce the program automatically.

3.3 CHOICE OF THE TARGET LANGUAGE

Designers may choose different languages for different tasks. A software development system should support this and permit the choice of any desired language as target language. At the same time it should make use of the fact that for each procedure a language hierarchy can be derived from the design hierarchy and, if the design is complete, the bottom layer of this hierarchy consists of the target language constructs. The system should be able to extract this language from the design and compile the whole design into it without requiring any special specification of the target language from the user.

Being a general design aid, such a system should not attempt to go any further. It should not produce machine code from the target language program (unless the target language itself is machine language) since this would imply the incorporation of a general compiler into it which has to be different for each facility and for each set of target languages to be used and would limit the target languages to the ones it can compile. One is far better off taking advantage of existing compilers to translate the code produced by the system into machine language.

3.4 PORTABLE AND ADAPTABLE PROGRAMS

A software system may be required to undergo various modifications during its lifetime for which there are basically two kinds of reasons: Changes in the environment in which the system has to run on one hand and changes in user requirements on the other. In the first case the measure of ease with which the system can be modified is called portability, in the second case it is called adaptability.

Changes in the environment can range from those which require only minor modifications - like replacing a compiler by a new one or moving the system to another computer, where the language dialect is different - to major changes like the need to use another language, possibly with different kinds of operations and data types, which might even require changes in the algorithms. Language cnanges are often part of the program development process. A program may be written in a high level language in order to check the feasibility of an idea, and then, in order to increase its efficiency, parts of it are rewritten in assembly language, or sometimes the entire program is transferred to a lower level and more efficient language, possibly on another machine. In future there will be languages and compilers that turn out instructions for creating actual hardware devices rather than code for an existing machine. This is another case where one might want to check a program in a high level language intending to write it in a different language ultimately.

User caused changes usually consist of deleting unnecessary features, adding new ones or improving the performance of certain parts of the system.

The main goal of the system discussed here is to support the creation of new programs. Contributing to the adaptability and portability of these programs certainly is an issue when building such a system.

4 EXISTING SOFTWARE DEVELOPMENT AIDS

The preceding chapter stated a list of objectives a software design system should meet. The goal of this chapter is to show how a system that meets these objectives relates to software development support systems which either exist at present or are likely to exist in the near future.

4.1 FROM METHODOLOGY TO ACTUAL SYSTEMS

In 1969 E.W. Dijkstra published an article called "Notes on Structured Programming" at the University of Eindhoven in the Netherlands. In 1971 N. Wirth from the Eidgenoessische Technische Hochschule in Zurich, Switzerland published an article called "Program Development by Stepwise Refinement" [38] in which he demonstrates the technique proposed by Dijkstra on the problem of the eight queens. Dijkstra's article was published in a book [10] in the year 1972. Several more editions of this book have been published since then. The program design technique advocated in these papers is that of dividing the problem to be solved into sub-problems, dividing each of the sub-problems into smaller sub problems and so on until each problem to be solved is simple enough to be treated as a whole. This methodology is referred to as "structured programming" or "structured design".

The structured programming technique was introduced into the US in the early seventies. According to articles and books published in those days it seems that there was a concensus about its advantages. In a book by E. Yourdon and L. Constantine [39] which was first published in 1975 the authors analyze the methodology and show that it leads to more efficiency both in the design process and in the resulting programs as well as better reliability, maintainability and generality.

There never was much argument about the question whether a design should proceed top-down, bottom-up or in any other way. The designs on which Dijkstra demonstrates the methodology proceed in an iterative top-down manner, but he never claims that this is the only way. Basili and Turner in [2] as well as Wilkes in [37] make the point that often, though one does not yet know exactly how the problem should be solved, he may know about subprograms his system will need. In such cases it may prove to be easier to start the design at those subparts and proceed from there in all directions.

As a result of the success of the structured design methodology and of the recognition that design is one of the major parts of the activity of producing a software system (Brooks in [5] estimates that one third of the development time is dedicated to it), software systems which support this activity have emerged since the mid seventies. Examples of such systems are PDL [6], SDDL [19], PSL/PSA [34], WELLMADE [41]. The objective of these systems is to support the activity of systematic design. The input consists of dynamic descriptions (algorithm descriptions) of modules which are stepwisely refined into sub-modules as well as, possibly, static and functional descriptions which may include details such as input and output specifications, data-types, names of people involved in the development, security levels etc. The system uses the input to build a data-base out of which documentation (video or hard-copy) may be provided. This documentation serves as a blueprint for programmers as well as a useful tool for project management.

To give the reader a better idea of the way these systems are used, here is a more detailed description of SDDL [19] (Software Design and Documentation Language) which was developed at the Jet Propulsion Laboratories in Pasadena, California. The input to SDDL consists of a series of module descriptions. The keywords PROGRAM, ENDPROGRAM, PROCEDURE, ENDPROCEDURE are used to identify the modules. Keywords like IF, ELSE, ENDIF, LOOP, CYCLE, ENDLOOP, CALL are used to describe the control flow of the algorithm within the modules. A third set of keywords like EJECT, IDENT etc is used to control the output formats. The output consists of two main parts. The first part includes all the modules printed in a nice and easy to read way where lines between keywords are indented and line numbers are supplied by the system. The second part consists of tables which are useful to get a quick overview of the system and of its interrelationships. One table lists the contents of design document by showing in terms of page and line numbers where the modules and the other tables are located. Cross reference tables show where words and module names used in the design are mentioned in the modules. Another table contains a module reference tree the inter-module calls.

The inputs and outputs of the other design systems are of similar nature with, possibly, some additions or deletions.

A further step in software development supporting software which is worth mentioning was the introduction of systems for verification of designs and of programs in the late seventies. Some are incorporated into the design systems described above, some are independent. Above mentioned PSL/PSA includes such a system. It provides reports about subjects like input/output consistency, gaps in information flow or unused data objects. Examples of independent systems are REVS [4], DISSECT [15] and EFFIGY [18]. REVS (Requirements Engineering and Validation Systems) includes a Requirement Statement Language (RSL) in which the data flow in the software system to be designed is described using a Requirement Statement Language (RSL). This description is used to build a relational database called Abstract System Semantic Module (ASSM). A set of programs is then used to analyze the database for completeness and consistency as well as to simulate the data flow through the model.

DISSECT and EFFIGY are examples of a different kind of verification systems. They check the program by performing a symbolic execution. This means that rather than returning a value they return the formula which the program computes.

To conclude this paragraph here are a few remarks concerning the design systems. First it should be noted that these systems have been used successfully. They are quite easy to use and reports show improvements in the amount of control of project managers over the activities in their groups as well as improved programmer productivity and less design and programming errors. An increasing number of major companies have been introducing design systems into their software development facilities and in doing so have saved considerable amounts of money.

A second remark about these systems is that none of them produce actual programs. When the design is completed the programmer has to write the program by hand using the design as a blueprint. In terms of the discussions in chapters 2 and 3 one may say that these systems do not distinguish between the design organization and the system organization. The macro-procedure decisions are made after completing the design rather than being incorporated into it. The automated coding objective discussed in chapter 3 states that the gap between the design and the program which currently is closed by hand, can - and should be - closed by the computer.

4.2 PROGRAM-PRODUCING SYSTEMS

The desirability of a system which produces a program from a design has been discussed in preceding paragraphs and chapters. It might be worth mentioning at this point that systems which produce programs, though not from designs, have been - and are being - developed.

An interesting research in this area has been conducted by J. Hobbs [14] first at the City University of New York and presently at Stanford Research Institute. A system that accepts a "well-written" algorithm description in a sublanguage of English and translates it into a PL/1 program is currently under development. It incorporates an existing system for the semantic analysis of texts in English (SATE). The algorithm will be first translated into a logic representation by the semantic analyzer, and from there - into PL/1. The semantic analyzer contains a lexicon where, associated with each word, is a collection of facts relating it to other entries. For example - the lexicon "knows" that a binary tree is a data-structure, that it is composed of nodes, that it has a root, and that the root is a node. The lexicon is used to transform the English description into logic representation. For example the sentence:

The variable points to the root of a binary tree

is transformed into:

```
point([x1|variable (x1)],
      [x2|root(x2) ,[x3|binary-tree (x3)]])])
```

Another example is a system being developed by R. Balzer at the USC Information Sciences Institute [1]. This system will accept a probelm and an algorithm description in a LISP-like format and perform transformations, most of which are automatic, on the input in order to translate it into a computer program.

Systems for automatic selection of library routines like [26] and [27] are another example of research in this field. These systems select representations and associated routines of commonly used data structures like stacks, queues and trees in order to maximize the efficiency of the programs using them. Information flow in the programs, sample runs and user interrogation are used to make the selection.

Systems like the ones desribed above will provide very attractive programming environments once they are operational. However, as already mentioned, none of them produce programs from the design. They all require a program which is very high level, but nevertheless has to include a description of the algorithm. These systems reduce the gap between the design and the program by providing a set of very high level primitives that can be used as target language of the design. They will form a nice complement to a design system which meets the objectives of chapter 3.

4.3 PROGRAMMING LANGUAGES

High level programming languages were developed in order to provide the user with a means of expressing an algorithm in a way that can be made to be understood by the machine and, at the same time, avoid the necessity of writing parts of the program which do not contribute to the description of the algorithm and whose mere function is to match it to prescribed machine features.

In reality it turns out that, for a large number of cases, a perfect match between a programming language and an application can not be found. This is a result of the inertia and rigidity of languages. In the development stages of a new language it is impossible to predict all the applications it might be used for and hence - all the demands it may be required to meet. On the other hand, once the development and production phases are completed, it is a difficult task to change the translator in order to match the language to new unforeseen applications. As a result, programmers often have to spend time trying to force existing languages to fit their needs. There are different ways to overcome this problem to some extent - each with its advantages and its drawbacks.

The idea of a universal language is one possibility. A language is universal if it can satisfy the needs of any programmer for whatever application he might use it. It has to provide data structures and operations in all possible fields like: Arithmetic, text processing, list processing, simulation etc. This implies the construction of a very large and complicated translator together with all the problems associated with such a system: It is difficult to produce and to maintain, it would require long translation times and, because of the difficulty in optimizing such a large system, the object code produced by it would often suffer from inefficiencies. Because of its size it would be unusable for many users who operate small machines. Further - since, as mentioned above, it is impossible to predict presently non-existing applications, it may well be, that today's universal language will not be so in the future.

On the other extreme the problem could be solved by using a large number of special purpose, problem oriented languages from which each user can pick the one that best suits his needs. In this way one can enjoy the advantages of a small system: It is relatively easy to write and to maintain, it runs faster and it is easier to optimize and be made to produce more efficient code. A drawback of this solution is the large number of different languages a computing facility has to keep. The efforts needed to maintain each language add up and make the maintenance of the whole facility costly both in terms of time and money. Again there is the problem of keeping up with developments and the necessity to provide new languages as new needs arise. Finally - the human factor: there is a phenomenon of "loyalty to the language". A programmer often has a small number of programming languages (in many cases only one language) he has used a lot, feels comfortable with and has to do only a small number of manual look-ups when he uses them. He is reluctant to learn a new language unless absolutely necessary and prefers to use his favorite language even if it does not fit his current problem too well. Therefore it could well be that out of a large number of languages a computing facility keeps, only a small subset is actually being used.

The emergence of extensible languages in the late sixties and early seventies was an attempt to tackle the problem of matching the language to the application. Extensive surveys and evaluations of these languages can be found in [28], [29], [30] and [32]. An extensible language consists of a fixed kernel called the base-language and an extension mechanism by which the kernel can be modified and/or extended. A program in an extensible language consists of definitions which extend the base language and instructions in the extended language which are then compiled or interpreted.

A number of different extension mechanisms can be found in these languages. One kind of such mechanism is the macro definition. Initially only assembly languages provided the ability to write macros, then it was recognized that macros can serve as powerful tools in high level programming languages as well [20,23].

One of the first high level languages into which a macro processor was incorporated was Algol [21]. Another common extension mechanism is the ability to define new operators in terms of old ones. For example ELF [7] MAD [11], and BALM [13] include such facilities. An example of a MAD definition is:

```
DEFINE BINARY OPERATOR .CONCAT.,
PRECEDENCE HIGHER THAN .ABS.
MODE STRUCTURE 1 = 1.CONCAT.1
```

The third line of this definition defines the datatypes on which the new operator works. Another, more general, definition mechanism is the ability to modify, delete or insert syntax rules and their semantics. IMP [16], ECT [31] and ECL [36] are such languages. An example of a rule modification in ECT is:

```
DELETE   F = V!C
ADD      P = V!C
```

This input specifies that the BNF syntax rules:

```
<F> ::= <V>
<F> ::= <C>
```

are to be replaced with the rules:

```
<P> ::= <V>
<P> ::= <C>
```

(see 5.5 for an explanation of the BNF notation)

The idea which led to the introduction of extensible languages was that a computing facility could keep a small number of extensible languages, such that the required maintenance efforts are not too large, and still satisfy the programmers since every user could use a language that fits his needs by extending one of the kernel languages. Yet, in spite of the advantages, it turns out that extensible languages are not widely used even in the computer-science community. Standish in [32] explains why this happened. Many kinds of extensions require modifications of the language processor which are not trivial and require skill and knowledge that many users do not - and are not expected to - possess. Therefore only trivial extensions, if any, were used, and the efforts of the language designers to provide a sophisticated environment were wasted. It seems to me that another reason for the rejection of extensible languages is that the fact that a language can be extended, even only superficially, confronts the user with a much larger decision space than a fixed language does. When using a fixed language with prescribed constructs the only decisions the user has to make are those concerning the method of solving his problem and the utilization of the language to perform the task. An extensible language introduces another dimension into the decision space. The user has to decide how to extend the language to fit his needs. These extra decisions require additional mental efforts and time and often become a burden rather than an advantage, a burden which may be heavier than the one introduced by the need to tailor a solution with a fixed language.

Most facilities have adopted a compromise between the first two ways described above. They keep a medium number of languages such that the maintenance costs do not run too high, and the user has some choice and can pick the language that most closely suits his application, taste or habit. This is the reason for the requirement (in 3.3) that a software design system should be able to translate the design into any target language. Only if this requirement is met will the system be useful for a large spectrum of users in different facilities.

4.4 TRANSLATORS

A major part of SDS, the software design system developed according to the objectives in chapter 3, is constituted by a language translator which accepts any grammar. Before discussing the SDS translator in later chapters, it may be worth while to aquaint the reader with some of the existing general translators.

Almost all the research and development efforts in the field of translators have been directed towards the creation of systems which translate programming languages into machine code, ie - compilers and interpreters. A comparison of two such translators will usually reveal that, though they may translate different languages and have different implementations, they use similar data structures (e.g. tables and stacks) and algorithms (e.g. for lexical analysis and parsing). This recognition led to the development of so called "translator generators" or "compiler compilers". These systems include programs and/or data structures which have been found to be common to many compilers or interpreters. Building a compiler from such a system is usually easier and faster than starting the work from scratch.

One of the first compiler compilers [22] called "The Compiler Compiler" was developed at the University of Manchester in England in the early sixties. It is used by writing the syntax rules of the language to be translated and their corresponding semantic routines which are written in assembly language. A built-in left-to-right parser will then process the input according to the rules entered and call the semantic routines. A number of compiler compilers which operate in a similar manner have been constructed since. A recent example is YACC (Yet Another Compiler Compiler) [40] which was developed at Bell Laboratories in Murray Hill, New Jersey. YACC consists of a parser and of a lexical analyzer. It accepts syntax - and lexical - rules and the corresponding semantic routines which are usually written in C Language. An example of input to YACC is:

```
NUMBER      : DIGIT
                        { $$ = $1; }
            ! NUMBER DIGIT
                        { $$ = 10 * $1 + $2; }
            ;
```

This input specifies the BNF rules:

```
<NUMBER> ::= <DIGIT>
<NUMBER> ::= <NUMBER> <DIGIT>
```

The semantic routine of the first rule returns the value of the digit. The semantic routine of the second rule returns the sum of the value of the digit and ten times the value of the number.

An example of a different kind is the TGS-II Translator Generator System [8]. It consists of various tables and associated programs a compiler writer may need such as tables for symbols, literals, terminal symbols, operation codes, labels. The user of this system writes all the translation phases himself, but is spared the effort of designing and implementing most of the data structures he may need.

4.5 PORTABILITY AND ADAPTABILITY AIDS

The straightforward way to modify a software system is to rewrite the program. This is, in most cases, also the most laborious way. Better portability can be achieved if the transition between the two languages would be automated. Then, after the initial effort of building a translator, a whole class of programs can be translated easily without spending time and effort for each program.

A solution along this line was proposed as early as 1958 [33]. The idea was to develop a language called: UNCOL (Universal Computer Oriented Language), which would serve as an intermediate level between any high level language and any machine language, and to build translators from all high level languages into UNCOL and from UNCOL into all machine languages. Then a program could be run on any machine after undergoing two translation phases. The reason why this idea was never implemented is the impracticality of constructing a universal language as described in 4.3 above.

Poole and Waite [24] developed a system that operates on similar principles but uses more than one possible intermediate level. The basic idea here is to define a set of abstract machines each of which suits a particular class of problems. All the programs written for such a machine can be translated into machine language by coding each operation of the abstract machine as macros in terms of the real machine and using a macro processor to do the translation. This system has been implemented on different computers without major difficulties.

5 GETTING ACQUAINTED WITH A SOFTWARE DEVELOPMENT SYSTEM

A software design system (SDS) that attempts to meet the objectives of chapter 3 has been designed and implemented. This chapter acquaints the reader with the system and its use. Detailed descriptions of the system and its implementation are in the following chapters and in the appendices.

5.1 THE GENERAL IDEA

The idea of the system is based on the notions discussed in the previous chapters. The process of design can be modelled as a progression through a series of conceptual languages. This was discussed in 2.4. Moving down this hierarchy one sees languages with increasing expressiveness. If design block S is refined into the sequence S1 S2 S3, then S belongs to a language L that is one step higher (and therefore one step lower in its expressiveness) than the language L' to which S1 S2 S3 belong. Further - the sequence S1 S2 S3 is the translation of S into L'.

The system lets the designer define the languages through which he progresses by specifying three things: (i) the syntax rules of the language constructs; (ii) how each construct should be translated, in other words: what does each construct mean in terms of the next lower language; (iii) whether the rule is a macro or a procedure. The rules so defined correspond, in general, to blocks of the design hierarchy. The rules are kept in a structure called the user's dictionary. A language processor is then used to perform a series of translations according to those rules and to produce code in terms of the lowest language called the Target Language.

The set of languages that the system goes through in producing the code is, in general, different from the sequence of the designer's conceptual languages. It is the macro - procedure distinction that provides the necessary information in order to extract these languages and to make the correct translation process possible.

In order to meet the flexibility requirement and let the user do the design in any order, the system works in two passes: (i) A syntax pass, in which the user's dictionary is constructed; (ii) a translation pass, in which the output code is compiled. The user has full control over these passes and can invoke them when desired.

5.2 SYNTAX DRIVEN LANGUAGE PROCESSING

In the previous paragraph I mentioned that the translation is performed by a language processor. Before proceeding with the description of the whole system here are a few words about the language processor's operation.

Languages are formally described by means of syntax rules and associated meanings (also called: semantics, or interpretations). These meanings are, in computer languages, actions (which will also be called semantic routines, or, in short semantics) to be performed whenever a string obtained by the corresponding rule is encountered. The dictionary of a language contains all its rules together with their associated meanings.

In order to process a string of characters that is supposed to belong to the language, one first has to parse it in order to verify that it is indeed part of the language and to find out which rules have been used to compose the string, and then, if it parsed successfully, to execute the corresponding semantic routines.

5.3 A SAMPLE INPUT TEXT

The following text (without the line numbers) is an example of an input to the system:

```
 1   PROC
 2   <STMT>: ADD ’A’ TO ’B’
 3   WHERE
 4   A,B: <ID>
 5   -->
 6   SET ’B’ TO SUM OF ’A’ AND ’B’
 7   PEND
 8
 9
10   MACRO
11   <STMT>: SET ’U’ TO ’V’
12   WHERE
13   U:<ID>  V:<EX>
14   -->
15   ’U’:=’V’
16   MEND
17
18
19   MACRO
20   <EX>: SUM OF ’X’ AND ’Y’
21   WHERE
22   X,Y: <EX>
23   -->
24   ’X’+’Y’
25   MEND
26
27
28   MACRO
29   <EX>: ’N’
30   WHERE
31   N:<ID>,<NU>
32   -->
33   PRIMITIVE
34   MEND
```

In the following paragraphs this example is used to illustrate the system’s operation. Macros and procedures are referred to by the line numbers at which they start. Only the basic operations are described here. More aspects will be discussed in later chapters.

5.4 DEFINITIONS, NOTATIONS AND SOME SYNTAX

A complete syntax of the system is given in appendix A. The part of this syntax which is necessary in order to follow the example and a few notations which are used in later discussions are given in an informal way in this paragraph.

The user's input consists of procedures and macros, both of which constitute design blocks as explained in chapter 2. Both procedures and macros provide refinements of higher level constructs into lower levels. Besides being refinements, the macro definitions are also used as translation rules which are applied to the procedures in order to produce the actual modules of the final program.

PROC and MACRO are keywords indicating the beginning of a procedure or a macro respectively. PEND and MEND indicate their end.

The part of a procedure or a macro that precedes the arrow is the <u>left-hand side</u> (lhs). Lines 1-4 constitute the left-hand side of the procedure at the beginning of the example; lines 10-13 constitute the left-hand side of the first macro.

The part that follows the arrow is the <u>right-hand side</u> (rhs). Lines 6 and 7 form the right-hand side of the procedure (also called: <u>Procedure body</u>); lines 15 and 16 are the right-hand side of the first macro.

The left-hand sides of both macros and procedures contain a text called lhs text whose refinement is given in the right-hand side by the rhs text. For example, the lhs text of the procedure (line 2):

```
ADD 'A' TO 'B'
```

is refined into the rhs text (line 6):

```
SET 'B' TO SUM OF 'A' AND 'B'
```

Names between single quotes, like 'A' and 'B' above, are parameters. They stand for any text of certain parts of speech (abbreviated: pos; also called: syntactic categories). The specification of the parts of speech for which each parameter stands is given in a declaration which follows the keyword WHERE. Line 4 in the procedure is such a declaration. It indicates that both A and B stand for texts of the syntactic category: <ID> (identifier). Thus if, for example, MAX, FLAG and P4 are identifiers and 8 is not, then the texts:

```
ADD MAX TO FLAG

ADD P4 TO MAX
```

mean, according to this procedure:

```
SET FLAG TO SUM OF MAX AND FLAG
```

and

```
SET MAX TO SUM OF P4 AND MAX
```

but the text

```
ADD 8 TO MAX
```

is meaningless with respect to this procedure because the requirement that 'A' has to be replaced by an identifier has not been met.

As the example shows, the left-hand side text is always preceded with a part of speech followed by a colon (like <STMT>: in lines 2 and 11; and <EX>: in lines 20 and 29). It will be referred to as the lhs pos. and it indicates the syntactic category to which the lhs text and its refinement (the rhs text) belong. Thus, according to lines 2 and 6, texts like:

```
ADD P4 TO MAX
```

and

```
SET MAX TO SUM OF P4 AND MAX
```

belong to the category <STMT> (statement); and, according to macro 19, if INDEX and 1 belong to the category <EX> (expression) then the texts:

SUM OF 1 AND INDEX

and

1+INDEX

are also expressions.

Macro 28 is of a different kind than the other macros of the example. Its rhs text (line 33) consists of the word PRIMITIVE. This is a keyword and it indicates that texts which match the lhs should not be translated since they are part of the target language. Macros of this kind will be referred to as primitive macros.

As mentioned above, names between angular brackets, like <STMT> and <EX> indicate parts of speech. Four parts of speech are pre-defined in the system: <STMT> (statement), <ID> (identifier), <NU> (number) and <BLANK> (blank characters). However the programmer may use freely any parts of speech of his choice. The mere introduction of a part of speech in a procedure or in a macro suffices to introduce it into the system. For example: the part of speech <EX> is introduced to the system when it is first mentioned in line 13.

5.5 DESCRIPTION OF SYNTAX RULES

Throughout this writeup syntax rules are written in Backus Naur Form (BNF). Non-terminal parts of speech are represented by names surrounded with angular brackets, like ‹EX› or ‹NU›. As I mentioned above, such a part of speech represents a set of text strings which belong to the corresponding syntactic category. These parts of speech are merely a tool for describing the syntax of a language, they do not appear in actual texts which belong to the language and therefore are also called "non-terminal parts of speech". Subscripts are used if identical parts of speech have to be distinguished between (eg. $‹EX›_1$ $‹EX›_2$ etc.).

It is often necessary to include actual characters of the language in a syntax rule. These characters can be looked upon as parts of speech which represent only themselves. They are also called "terminal parts of speech" and are distinguished from non-terminal parts of speech by having no brackets around them. For example the sequence:

‹ID›:=‹EX›

consists of two non-terminal parts of speech: ‹ID› and ‹EX› and two terminal parts of speech: The characters ':' and '='.

The symbol ::= is used to separate the left-hand side of a syntax rule from its right-hand side. It stands for: "May be rewritten as". The entire rule states that the texts which are defined in the right-hand side belong to the syntactic category of the left-hand side. For example, the rule:

<STMT> ::= <ID>:=<EX>

says that text strings which consist of an identifier followed by a colon, followed by an equal sign, followed by an expression, belong to the category "statement" (represented by the pos <STMT>).

5.6 THE SYNTAX PASS

In the syntax pass the user's dictionary is constructed from syntax rules which are defined in macros and procedures.

Macros whose rhs is the word PRIMITIVE introduce one set of new syntax rules into the user's dictionary: Primitive lhs rules which will also be referred to as: Declared primitives, or as: Defined primitives.

Lhs rules have the lhs part of speech as their left-hand side. Their right-hand sides consist of the lhs text in which the formal parameters have been replaced by all possible combinations of parts of speech as found in the declarations. Each combination defines a new rule.

For example, in macro 28 (whose rhs is PRIMITIVE) there is one formal parameter 'N' which, according to the declaration in line 31, stands for two parts of speech: <ID> (identifier) and <NU> (number). The lhs part of speech (on line 29) is <EX>. This macro introduces the two rules:

```
<EX> ::= <ID>
```

and

```
<EX> ::= <NU>
```

These rules are primitives, ie - they are parts of the target language and therefore their semantic interpretation says: Leave the text that parsed according to them as it is.

Macros whose rhs is not the word PRIMITIVE introduce two sets of rules: Lhs rules and rhs rules. The lhs rules are obtained in the same way as in primitive macros. The semantic interpretation of a lhs rule in a non primitive macro is the substitution of the corresponding text with the rhs.

In macro 10, for example, there are two parameters: 'U' and 'V' which stand for the parts of speech <ID> and <EX> respectively. This macro introduces the lhs rule:

```
<STMT> ::= SET <ID> TO <EX>
```

If Y is an identifier and X+7 is an expression, then the text:

```
SET Y TO X+7
```

parses by this rule, hence the semantics will replace it, according to the form on line 15, with:

Y:=X+7

Rhs rules, like lhs rules, have the lhs part of speech as their left hand side. Their right hand sides consist of the rhs text in which the formal parameters have been replaced by all possible combinations of parts of speech from the declarations. Here again each different combination defines a rule. Macro 10 introduces the rhs rule:

<STMT> ::= <ID>:=<EX>

This rule was obtained by placing the part of speech <STMT> from line 11 on its lhs, and placing the form of line 15 (after replacing the parameters 'U' and 'V' by <ID> and <EX> respectively) on its rhs.

Rhs rules, as long as they are not redefined in another macro or procedure, are considered to be primitives, ie - parts of the target language. The translation system would also work correctly if these rules were not in the dictionary. However, as chapter 6 will explain in detail, there are advantages in keeping them. For reasons which too will be discussed later (in chapters 6 and 7), primitives introduced as rhs rules have to be distinguished from primitives introduced via primitive macros (like macro 28) as lhs rules. The distinction is made by marking the two kinds of primitives differently in the dictionary. I will distinguish between them in this writeup by referring to primitives introduced via primitive macros as <u>defined primitives</u> (or, in short, just: Primitives) and to primitives introduced as rhs rules as <u>implied primitives</u> (because the fact that they are primitives is implied by their status of rhs rules).

A special case of a non-primitive macro is a macro with an empty rhs. Such a macro introduces lhs rules whose semantic interpretation is a substitution with an empty string, ie - omitting the text that parsed by one of these rules. There are no new rhs rules introduced in this case.

Left hand sides of procedures are dealt with exactly like left hand sides of primitive macros. Each procedure introduces a set of defined primitives. Procedure 1, for example, introduces the defined primitive rule:

```
<STMT> ::= ADD <ID> TO <ID>
```

The first condition for a successful translation is that the text will parse. The introduction of this rule enables the designer to use the corresponding construct in any part (macros or procedures) of his design in order to indicate a call of the procedure. If, for example, BASE and DISPLACEMENT are identifiers, then the text:

```
ADD DISPLACEMENT TO BASE
```

can be used as part of any procedure or macro without interfering with a successful parse.

Right hand sides of procedures are ignored in the syntax pass.

In summary, here is a list of the rules that are introduced by the procedure and macros in the example. Each rule is followed by its semantic interpretation:

PROC 1:

(I) ⟨STMT⟩ ::= ADD ⟨ID⟩ TO ⟨ID⟩
Primitive

MACRO 10:

(II) ⟨STMT⟩ ::= SET ⟨ID⟩ TO ⟨EX⟩
Subst: ⟨ID⟩:=⟨EX⟩

(III) ⟨STMT⟩ ::= ⟨ID⟩:=⟨EX⟩
Implied primitive

MACRO 19:

(IV) ⟨EX⟩ ::= SUM OF $\langle \text{EX} \rangle_1$ AND $\langle \text{EX} \rangle_2$
Subst: $\langle \text{EX} \rangle_1 + \langle \text{EX} \rangle_2$

(V) ⟨EX⟩ ::= ⟨EX⟩+⟨EX⟩
Implied primitive

MACRO 28:

(VI) ⟨EX⟩ ::= ⟨ID⟩
Primitive

(VII) ⟨EX⟩ ::= ⟨NU⟩
Primitive

5.7 THE TRANSLATION PASS

In the translation pass procedure bodies (right-hand sides of procedures) are translated, using the translation rules that have been stored in the dictionary in syntax pass. The process will be described on procedure 1 from the example.

First the procedure is parsed, yielding the following parsing graph:

```
                       <STMT> (II)
|----------------------------------------------------------|
|                                                          |

                                    <EX> (IV)
                       |-----------------------------------|
                       |                                   |

                                     <EX>          <EX>
                                     (VI)          (VI)
                                    |-----|       |-----|
                                    |     |       |     |

        <ID>                         <ID>          <ID>
       |-----|                      |-----|       |-----|
       |     |                      |     |       |     |
 S E T  ' B '   T O   S U M   O F   ' A '    AND   ' B '
```

The arcs around the parameters 'A' and 'B' were built in a pre-parsing process according to the part of speech declarations in the lhs of the procedure. All the other arcs correspond to rules from the user's dictionary. The number in parentheses following each part of speech indicates the appropriate rule.

All the semantic routines used in translation are functions which return texts. Semantics of characters (leaves of the parsing tree) simply return the character. Semantics of parameters (like 'A' and 'B') return the parameter. Every other semantic routine first calls the semantics of its constituents and then, if it is a primitive, simply returns the concatenation of their outputs, otherwise it returns the translation in terms of the constituents outputs.

Since there exists a complete parse (there is an arc that spans the entire text) the semantic routine of the spanning arc is called. This routine calls the routines of the constituents, and the procedure calls bubble down to the leaves where they start to return. The parameters return the texts 'A' and 'B'. The semantic of rule IV is a substitution, consequently the arc marked <EX> (IV) returns the text: 'A'+'B'. The semantic of the <STMT> is a substitution that returns the text 'B':='A'+'B'.

Here the first translation iteration is completed. Since the output text is different from the input, the translation process is repeated. The output text is parsed resulting in the following parsing graph:

```
              <STMT> (III)
|-----------------------------------|
|                                   |

                     <EX> (V)
               |--------------------|
               |                    |

              <EX> (VI)   <EX> (VI)
               |-------|   |-------|
               |       |   |       |

  <ID>           <ID>        <ID>
|-------|      |-------|   |-------|
|       |      |       |   |       |
 '  B  '  : = '  A  '  + '  B  '
```

Here, again, the arcs around the parameters were obtained by a pre-processor. The semantics of these arcs return the parameters as they are. All the other arcs correspond to rules from the user's dictionary. All the rules, in this case, are primitives, therefore the output text is identical to the input and the tranaslation iterations are terminated.

As a last step the text is scanned and the quotes around the parameters are removed, yielding the text: B:=A+B.

6 MORE ABOUT SYNTAX RULES

6.1 DEFINED PRIMITIVES

Syntax rules can be explicitly defined as primitives of the target language. This can be done in two ways: Through primitive macros and through procedure definitions.

The first condition for a successful translation of a text is that it will parse. Therefore the dictionary must include syntax rules for all language constructs used in the text. Usually these rules are defined in macros (either explicitly or as implied primitives) or in procedure definitions. However the designer may want to incorporate into his design constructs which should not be refined into lower levels and hence do not occur in any refining macro or procedure. The primitive macro is a tool by which such rules can be introduced into the dictionary.

There are several cases in which one may need this facility. Often the designer may want to mix target language constructs together with higher level constructs which have to be translated. For example he may want to include in a procedure the statement:

```
SET X TO A+B
```

Suppose that X, A and B parse to <ID>. Further assume that the rule

`<STMT> ::= SET <ID> TO <EX>`

has been introduced by a non-primitive macro and should be sustituted with

`<ID>:=<EX>`

and that the rule

`<EX> ::= <ID>+<ID>`

is part of the target language.

To make the entire text parse successfully the last rule has to be in the dictionary so that A+B will parse to <EX>. The way to make this happen is to write a primitive macro which introduces this rule as a defined primitive. In fact a reasonable way to use SDS is to create dictionaries which contain the rules of frequently used programming languages as defined primitives, and to use the dictionary for a programming language as a starting point for all designs having it as a target language. In this way one can freely mix target language constructs with his own language constructs in the design.

Another case in which one may have to define a primitive is when he wants to use a high level, non-primitive, construct in a procedure, but does not want, for the moment, to worry about the refinement of that construct and yet wants to see how the rest of the procedure translates. Here again the whole procedure has to parse. This can be achieved if the high level construct is temporarily defined as a primitive by a primitive macro. Then, if there are no errors, the procedure will parse and be translated into a target language which, temporarily, includes this defined primitive. Later the primitive can be overridden by defining it with another, non-primitive, macro.

A third reason for writing a primitive macro is to override an implied primitive rule, which has lower priorities in both syntax and translation passes (see 6.3 and 7.4), by a defined primitive, whose priority in translation is the same as that of non-primitive rules.

The second way in which defined primitives are introduced is the lhs of procedures. Both procedures and macros provide refinements of their left-hand sides in terms of lower languages, but when a text parses according to the lhs of a procedure then, unlike the lhs of a macro, it is not substituted with the procedure's rhs text. It is regarded as the calling sequence of the procedure and considered part of the target language just like a defined primitive. Like any other primitive the rule can be re-defined via a macro in order to translate the calling sequence used in the design into the actual calling sequence used in the target language. For example, the procedure in the preceding chapter introduced the primitive rule:

```
<STMT> ::= ADD <ID> TO <ID>
```

Therefore, if X and Y are identifiers, the text

```
ADD X TO Y
```

may be used as part of any procedure or macro. If the target language is, say, Fortran, then this construct has to be translated into a proper Fortran subroutine call, so the programmer may add the macro

```
MACRO
<STMT>: ADD 'U' TO 'V'
WHERE
U,V: <ID>
-->
ADDTO('U', 'V')
MEND
```

Which changes the rule into the non-primitive:

$\langle STMT\rangle$: ADD $\langle ID\rangle_1$ TO $\langle ID\rangle_2$

Subst: ADDTO($\langle ID\rangle_1$, $\langle ID\rangle_2$)

6.2 IMPLIED PRIMITIVES

As explained in the preceding chapter, an implied primitive is a rule which describes the language construct used in the right-hand side of a macro to refine the text of its left-hand side.

The main reason for storing implied primitives is to enable the user to check the current status of his design. At any stage he may, from the command level of the system, ask what the primitives are and thus check what parts of his program are still undefined in terms of the target language. Storing the implied primitives in the user's dictionary, together with all other rules has two advantages: Once an implied primitive is in the dictionary, it participates in the parsing process, therefore there usually is no need to write a special primitive macro in order to insert the rule. The second advantage is that the process of overriding an implied primitive with a defined primitive or with a non-primitive rule is simplified in this way - all it involves is changing the type of the rule in the user's dictionary, whereas if it were kept in a different dictionary, every insertion of a defined primitive or a non-primitive into the user's dictionary would involve a search of the implied primitives dictionary in order to remove the implied primitive if it is found.

6.3 ORDERS OF PRIORITY

The user's dictionary contains all the syntax rules that the user defines. This includes non-primitive rules as well as defined primitives and implied primitives.

As mentioned in previous paragraphs, existing rules may be redefined and their types changed. These changes are subject to the following order of priority:

(i) Non primitives

(ii) Defined primitives

(iii) Implied primitives

This means that if an attempt is made to insert a rule that already exists in the dictionary and the new rule is of a type that has a higher priority than that of the old one, then the type of the old rule is changed to that of the new rule (which is equivalent to replacing the rule), otherwise no change is made. Further, if there is an attempt to re-define a non-primitive rule and the new rule has different semantics (ie - two macros define the same construct in different ways), the second macro is completely ignored (no rhs rules inserted either) and an error message is issued.

The priority of non-primitives over implied primitives reflects the idea of the development system. There is a hierarchy of languages through which the designer progresses. At any stage of the design there is a set of rules which have not been defined in terms of others. These rules form the current target language. If one moves down the hierarchy, he refines primitive constructs by defining them in terms of a lower language and they cease to be primitives. If he moves up the hierarchy, language constructs, which have already been defined in terms of others and thus are not primitives, are used in right-hand sides of macros to define parts of a higher language. Being rhs rules, the system attempts to insert them into the dictionary as implied primitives, but this attempt has to fail.

Non-primitives have priority over defined primitives since, as discussed above, the designer may want to refine primitives which were introduced only temporarily or primitives which are used as procedure calls.

The reason for the priority of defined primitives over implied primitives is that implied primitives have lower priority in the translation process (see 7.4). If the user wants to override this inferiority and specify that the rule should be treated as equal to non-implied rules, he can do so by defining it in a primitive macro which results in removing the implied-status from the rule.

7. PICKING THE PARSING TREE

7.1 STATEMENT OF THE PROBLEM

The parser used in the system can handle any general rewrite rule grammar. It builds all the possible arcs around the input string and, with a reasonably sized syntax, the resulting parsing graph consists of a large number of arcs, most of them spurious, some of them desired. If at least one arc spans the entire input string, then the parsing graph is said to contain a complete parse and the act of parsing is said to be successful.

Any spanning arc is the root of a tree whose leaves are all the initial arcs of the input string. The semantic evaluation starts at the root of a parsing tree and recurses down to the leaves. In order to translate the text there has to be at least one parsing tree, because otherwise, though parts of the text may have their translations, the input string as a whole is semantically meaningless.

For reasons which will be clarified later in the chapter, many input strings, which parse successfully, end up with several spanning arcs, each corresponding to a different parsing tree and hence a different translation. The user usually has only one translation in mind for a given input, and it is the task of the translation system to resolve the ambiguities and to find the corrrect parsing tree.

Ambiguous parsing graphs can be divided into four categories, two of which are rather trivial and are dealt with in the next paragraph. The other two categories require a more complicated algorithm which is explained in the rest of the chapter.

7.2 TWO TRIVIAL CASES

The first category of ambiguous parsing graphs is where the ambiguity is a result of syntax rules whose right hand side contains a single part of speech. Such rules may produce parsing graphs which have more than one spanning arc and therefore look ambiguous but actually are not ambiguous at all. The following example clarifies this case.

Suppose that one wishes to take a list of statements which are separated by blanks and to insert a semicolon after each of the statements. A way to do it is to write the following macros:

```
MACRO
<STMTL>:'S'
WHERE
S:<STMT>
-->
'S';
MEND

MACRO
<STMTL>:'SL' 'S'
WHERE
SL:<STMTL>  S:<STMT>
-->
'SL' 'S';
MEND
```

Here STMT and STMTL stand for statement and statement-list respectively. These macros introduce the following non-primitive syntax rules:

```
<STMTL> ::= <STMT>
    Subst: <STMT>;

<STMTL> ::= <STMTL> <STMT>
    Subst: <STMTL> <STMT>;
```

Let S1 S2 and S3 parse to <STMT>. The processing of two input texts will be demonstrated, one of which has a non-ambiguous parsing praph, the second has a seemingly ambiguous parsing graph. The similarity between the two will show that the ambiguity in the second case is only an "optical illusion".

The first input text is: S1 S2 S3. It yields the following parsing graph (spurious arcs which do not contribute to a spanning arc have been omitted):

```
         <STMTL>
¦-------------------------¦
¦                         ¦

     <STMTL>
¦---------------¦
¦               ¦

<STMTL>
¦-----¦
¦     ¦

<STMT>    <STMT>    <STMT>
¦-----¦   ¦-----¦   ¦-----¦
¦     ¦   ¦     ¦   ¦     ¦
 S   1     S   2     S   3
```

There is exactly one spanning arc in this graph. Picking this arc as the root of the parsing tree and evaluating the semantics result in the desired output: S1; S2; S3; .

The second input text is: S1. The corresponding parsing graph is:

```
             <STMTL>
             ¦-----¦
             ¦     ¦

             <STMT>
             ¦-----¦
             ¦     ¦
              S   1
```

The graph has two spanning arcs. This is a result of the fact that the first syntax rule happens to have only one part of speech on its right hand side. The correct root is, just as in the previous example, the arc <STMTL>, and the resulting output text is: S1; .

Another way to look at these rules is to view their right-hand sides as having two parts of speech one of which is <EMPTY> to which every empty text parses and whose semantics return an empty string. The first rule can be written as:

<STMTL> ::= <STMT><EMPTY>

Using this rule the parsing graph of S1 is:

```
         <STMTL>
   |----------------|
   |                |

<STMT>       <EMPTY>
|-----|      |-----|
|     |      |     |
 S    1
```

In this way the optical illusion is gone and there is only one spanning arc left.

To summarize: rules whose right-hand sides consist of only one part of speech carry semantic meaning just like any other rule and are inserted into the dictionary in order to be applied whenever an input string parses to them. The arc that has to be picked as the root of the parsing tree in such a case is the one that covers the maximum number of arcs, ie - the highest spanning arc. In the rest of the chapter any mention of a spanning arc will refer to the highest.

The second category of ambiguous parsing graphs is where there are two or more spanning arcs with different parts of speech. This implies that the input text means two different things in terms of the current language that the designer is using and can not be resolved. It is regarded as a user error, the translation process is aborted and an error message stating that the text is ambiguous is issued.

Note that subtexts may have more than one spanning arc with different parts of speech as long as the text as a whole parses only to one part of speech. In such a case the ambiguity is internal, it is not reflected externally because the context in which the ambiguous subtext appears disambiguates it (all the arcs except the correct one become spurious). This kind of ambiguity can serve as a useful tool for the designer who wants to use similar constructs in different places and make the context determine the correct translation. For example in Pascal [17] one may need an ambiguous type which one may get in the cumbersome way of declaring a record with a variant attribute whose name has to be different for each type. In an SDS design one could use the same name for all cases and let the context in which it appears disambiguate it.

7.3 USING AMBIGUITIES FOR SPECIAL CASES

The third category of ambiguous parses is the one where the parsing graph has several spanning arcs - all with the same part of speech. One way in which this may occur is when the designer, usually for reasons of efficiency and optimization, specifies a special translation rule for a text that otherwise would fall in a more general category and would be translated differently.

For example - let the target language be an assembly language which includes the instructions:

ADDI N	(add the number N to the contents of the accumulator)
INC	(add 1 to the contents of the accumulator)
LOAD M	(move contents of location M into the accumulator)
STORE M	(move contents of accumulator into location M)

Suppose that the INC instruction is more efficient than the ADDI instruction. In order to translate statements like: Y:=X+5 into the target language one may write the macro:

```
MACRO
<STMT>: 'B':='A'+'N'
WHERE
A,B:<ID>  N:<NU>
-->
LOAD  'A'
ADDI  'N'
STORE 'B'
MEND
```

For efficiency the user might want to use INC instead of ADDI whenever the number following the '+' is 1. So he may write a special macro for this case:

```
MACRO
<STMT>: 'B':='A'+1
WHERE
```

```
A,B:<ID>
-->
LOAD  'A'
INC
STORE 'B'
MEND
```

The non-primitive rules inserted into the dictionary are:

```
(I)  <STMT> ::= <ID>:=<ID>+<NU>
       Subst: LOAD  <ID>
              ADDI  <NU>
              STORE <ID>


(II) <STMT> ::= <ID>:=<ID>+1
       Subst: LOAD  <ID>
              INC
              STORE <ID>
```

Suppose that X and Y parse to <ID> and every number parses to <NU>. A text like: Y:=X+8 (where the number is not 1) parses by the first rule only and will be translated correctly, using the ADDI instruction. The text Y:=X+1 parses ambiguously. The parsing graph, in this case, has two spanning arcs with the part of speech <STMT> and hence includes the two following parsing trees:

```
       <STMT> (I)                         <STMT> (II)
|-----------------------|          |-----------------------|
|                       |          |                       |

<ID>        <ID>    <NU>           <ID>        <ID>
|---|       |---|   |---|          |---|       |---|
|   |       |   |   |   |          |   |       |   |
  Y   :   =   X   +   1              Y   :   =   X   +   1
```

The tree that has to be picked in order to match the user's intention is the one that parsed according to rule II whose semantic uses INC rather then ADD.

Here is another example:

The user's dictionary includes the following rules:

```
(I)    <COND> ::= <ID><=<ID>
           Primitive

(II)   <COND> ::= <ID>>=<ID>
           Primitive

(III)  <COND> ::= <COND> AND <COND>
           Primitive
```

A special case is defined in the macro:

```
MACRO
<COND>: 'A'<='B' AND 'A'>='B'
WHERE
A,B:<ID>
-->
'A'='B'
MEND
```

which introduces the non-primitive rule:

```
(IV)  <COND> ::= <ID><=<ID> AND <ID>>=<ID>
            Subst: <ID>=<ID>
```

If X and Y both parse to <ID> then the text: X<=Y AND X>=Y has two possible parsing trees:

```
                         <COND> (III)
|----------------------------------------------------|
|                                                    |

   <COND> (I)                          <COND> (II)
|---------------|                   |---------------|
|               |                   |               |

<ID>        <ID>                    <ID>        <ID>
|---|       |---|                   |---|       |---|
|   |       |   |                   |   |       |   |
  X   <   =   Y       A   N   D       X   >   =   Y
```

and:

```
                         <COND> (IV)
|----------------------------------------------------|
|                                                    |

<ID>        <ID>                    <ID>        <ID>
|---|       |---|                   |---|       |---|
|   |       |   |                   |   |       |   |
  X   <   =   Y       A   N   D       X   >   =   Y
```

In the first tree all the rules used are primitives hence the translation is identical to the input: X<=Y AND X>=Y . In the second tree non-primitive rule IV is used and the translation is: X=Y , which the user had in mind for this case.

In both examples it turns out that the correct choice is that of the parsing tree which has less branches. This is not a coincidence, it is typical for special case definitions. Right-hand sides of syntax rules may contain parts of speech and individual characters. A part of speech is an abstraction. It represents a collection of texts that parse to it. This is what enables the representation of the large (often infinite) set of all the legal strings of a language by a finite, relatively small, number of rules.

In a parsing tree the root represents the highest level of abstraction. Knowing the part of speech of the root one usually does not know what the leaves are. They may be any text from the set that the root represents. Moving along the branches from the root to the leaves, the level of abstraction decreases. Each step through a branch corresponds to reducing the set of possibilities to a subset of the set of possibilities known so far.

In a special case definition one wants a subset of the set of strings represented by a part of speech to be treated differently than the rest of the set. The desired subset can be distinguished from the rest only by replacing the part of speech that represents the whole set in the syntax rule by an explicit definition of the subset. In the first example the part of speech <NU> that represents the set of whole numbers was replaced by the subset {1}. In the second example the two parts of speech <COND> in the rhs of rule III, which stand for the set of conditions, were replaced by certain subsets of conditions in rule IV.

Replacing a part of speech by an explicit subset corresponds to skipping one or more levels of abstraction, which results in a smaller number of branches in the parsing tree in which the special rule is applied.

The translation system makes use of these results. In case of multiple spanning arcs with identical parts of speech the number of branches in each parsing tree is counted and the one with the least number is picked.

If there are several trees with the least number of branches, this could be a result of applying different special cases or, maybe, a user error. For example, if the user introduced two special case rules:

```
<STMT> ::= <ID>:=<NU>+1
```

and

```
<STMT> ::= <ID>:=1+<NU>
```

then the text: Y:=1+1 would have two parsing trees, both of them resulting in a good translation.

On the other hand, due to a user error the syntax might permit ambiguous texts like:

```
IF C1 THEN IF C2 THEN S1 ELSE S2
```

where it is unclear whether to execute S2 if C1 is false or if C1 is true and C2 is false.

In cases of several minimal parsing trees, one tree is picked arbitrarily and a warning is issued to alert the designer to the possibility of an error.

7.4 AMBIGUITIES INTRODUCED BY IMPLIED PRIMITIVES

As mentioned in chapter 6, implied primitives are stored in the user's dictionary and stay there as implied primitives until their status as implied primitives is changed by re-defining them. This is true if the whole rule is explicitly defined in another macro. But there is another way to re-define primitives, namely by re-defining parts of them. In such a case the primitive rule stays in the dictionary as a primitve despite the fact that a text that parses according to it has to be translated. Here is an example:

Let the target language be ALGOL. The high level text: AVERAGE OF A AND B can be refined, as a first step, via the following macro:

```
MACRO
<EX>: AVERAGE OF 'X' AND 'Y'
WHERE
X,Y:<ID>
-->
HALF OF SUM('X' 'Y')
MEND
```

This introduces the rules:

```
(I)     <EX> ::= AVERAGE OF <ID> AND <ID>
           Subst: HALF OF SUM(<ID> <ID>)

(II)    <EX> ::= HALF OF SUM(<ID> <ID>)
           Implied primitive
```

One way to proceed from here is to re-define rule II with the macro:

```
MACRO
<EX>: HALF OF SUM('X' 'Y')
WHERE
X,Y:<ID>
-->
('X'+'Y')/2
MEND
```

The effect of this macro is to change the type of the implied primitive to non-primitive:

```
(III)  <EX> ::= HALF OF SUM(<ID> <ID>)
          Subst: ((ID)+(ID))/2
```

and to insert a new implied primitive:

```
(IV)   <EX> ::= ((ID)+(ID))/2
```

A second way to proceed is to replace the last macro by two macros. The construct: HALF OF SUM(<ID> <ID>) includes two high level expressions: HALF and SUM. They happen to be used together in this example, but the user may also want to use each of them separately or, at least, anticipate this possibility and therefore define each of them separately in the following macros:

```
MACRO
<EX>: HALF OF 'E'
WHERE
E:<EX>
-->
('E')/2
MEND


MACRO
<EX>: SUM('A' 'B')
WHERE
A,B:<ID>
-->
'A'+'B'
MEND
```

The corresponding syntax rules are:

```
(V)     <EX> ::= HALF OF <EX>
          Subst: (<EX>)/2

(VI)    <EX> ::= (<EX>)/2
          Implied primitive

(VII)   <EX> ::= SUM(<ID> <ID>)
          Subst: <ID>+<ID>
```

(VIII) <EX> ::= <ID>+<ID>
Implied primitive

Note that in this case the implied primitive rule II does not disappear from the dictionary.

Now follow the translation of the input text: AVERAGE OF A AND B,assuming that A and B parse to <ID>. In the first iteration the input text parses by rule I and translates into: HALF OF SUM(A B). The parsing graph in the second iteration depends on the way the user chose. If he picked the single-macro way, then the dictionary contains only rules I, III and IV, and the parsing graph is:

```
                         <EX> (III)
|-----------------------------------------------------------|
|                                                           |

                                                <ID>    <ID>
                                                |---|   |---|
                                                |   |   |   |

 H   A   L   F     O   F     S   U   M   (   A       B   )
```

There is no ambiguity here and, according to the semantics of rule III, the output is: (A+B)/2 .

If the user picked the two-macro way then the dictionary contains the rules I, II, IV - VIII which give rise to a parsing graph with two spanning arcs, both with the part of speech <EX>. The two parsing trees are

```
                        <EX> (V)
|---------------------------------------------------------|
|                                                         |

                                        <EX> (VII)
                              |---------------------------|
                              |                           |

                                            <ID>   <ID>
                                            |---|  |---|
                                            |   |  |   |
  H   A   L   F     O   F     S   U   M   (   A      B   )
```

and:

```
                        <EX> (II)
|---------------------------------------------------------|
|                                                         |

                                            <ID>   <ID>
                                            |---|  |---|
                                            |   |  |   |
  H   A   L   F     O   F     S   U   M   (   A      B   )
```

If the translation proceeds according to the first tree, the semantics of rules V and VII provide the correct translation: (A+B)/2 . If the translation proceeds according to the second tree, the text will not change, since rule II is a primitive hence the output will be: HALF OF SUM(A B) .

If the algorithm that picks the parsing tree were as described at the end of paragraph 7.3, then the second tree, which has less branches, would be picked, resulting in the wrong translation.

The problem obviously is a result of the fact that the implied primitive rule II was not re-defined as a whole and therefore was not removed from the dictionary although it ceased to be a primitive.

The system provides two solutions to this problem. One is incorporated in the algorithm that picks the parsing tree, the second is the UPDATE command that the user can issue from the command language level.

The algorithm that picks the root of the parsing tree starts by counting the number of implied primitive rules that were applied in each tree. All the trees in which the number of implied primitives is larger than the minimum number found are eliminated from further consideration. Then the process proceeds as describrd in 7.3 - a tree with the minimum number of branches is picked from the remaining set.

The need to count the number of implied primitives follows from the fact that a procedure may include several text segments which parse according to indirectly re-defined implied primitives. For each application of such a rule in a parsing tree there is another parsing tree in which the rule is not applied. Picking a tree with the minimal number of implied primitives applications assures that it contains no application of an indirectly re-defined rule.

The UPDATE command results in updating the dictionary by removing all the implied primitives that have indirectly been re-defined. In order to detect whether an implied primitive has been re-defined, its right-hand side is parsed (see also appendix A).

Parsing all the implied primitives in the dictionary can be time consuming, so the user may wish to avoid using UPDATE whenever he re-defines, or thinks that he re-defines, an implied primitive indirectly. In a typical design session the designer would write macros and procedures, mostly translate procedures without updating the dictionary, letting the root picking algorithm do the work, and once in a while or perhaps only at the end of the session, clean up the dictionary via the UPDATE command.

8 SEPARATION OF LANGUAGES

8.1 ONE DICTIONARY FOR ALL LANGUAGES

When the user designs his program, he defines a set of different languages. During the translation process the system proceeds through these languages step by step, translating the input text into the next lower language, translating the result into the next lower language and so on until the lowest level - the target language - is reached. At first sight it might seem that, in order to obtain successful translations, these languages have to be kept in separate dictionaries, and that the translator should move from one dictionary to another as the translation proceeds through the sequence of languages. However, fortunately, it turns out, that if certain restrictions are observed, such a physical separation is not necessary.

Why is it desirable to keep all languages in one dictionary? This is a result of the flexibility requirements. One requirement is that the designer should be able to proceed in any direction: top - down, bottom - up or any combination of them. This implies, that macros and procedures may be written in any order that pleases the user. Suppose he writes a macro where both left-hand side and right-hand side rules do not yet appear in any other macro or procedure and hence they do not appear in any dictionary. Then there is no way to tell where in the language hierarchy these rules belong unless explicitly specified by the user himself.

Another aspect of flexibility of design is that the user may use constructs from more than one language in procedure texts. If the language dictionaries were separated, then some translation steps might have to use two or more dictionaries simultaneously, and it is the designer who has to tell which of the dictionaries should be used.

A third aspect of flexibility is the ability to skip language levels. One may translate some high level constructs into the target language with fewer intermediate steps than other constructs. Thus the fact that, say, the left-hand side rule of a macro exists already in one dictionary does not necessarily imply that the right-hand side rule belongs in the immediately following dictionary. Here again the user's instructions would be needed in order to tell into which dictionary the rules should be inserted.

All the above examples have the same implication: The combination of flexibility on one hand and physical language separation on the other requires the user's awareness of the structure of the language hierarchy that he builds in his design, and his help both in maintaining this hierarchy and in using it correctly. But doing such a bookkeeping job is the last thing a software designer wants, and should have to do. The purpose of using a development system is to be able to avoid distracting and time consuming activities that are not part of the design process. The method in which the development system operates should certainly not be a cause for an excessive burden on the user. The user may, for example, find it natural to mix target language constructs together with high level constructs; he also may be unaware of the fact that a macro skips a language level, and, since he is only interested in producing a correct and working program, all other details do not, and should not, concern him.

8.2 LANGUAGE SEPARATION THROUGH PARTS OF SPEECH

Assuming that a text (a string of terminal parts of speech) belongs only to one language, if the sets of syntax rules of all the languages were mutually exclusive, then keeping only one dictionary would be no reason for confusion. A given text would parse by the rules of one language only and translate accordingly. The fact that rules from other languages are also present in the dictionary would have no influence on the translation process. But the real situation is sometimes different. Two or more languages may share structures that can be described with identical syntax rules, but, since the languages are on different levels, these structures have to be translated differently. Of course the assumption is that a text that may belong to more than one language, always appears in conjunction with some other text so that the context uniquely specifies the language. Otherwise there is no way to tell how it should be translated.

If two identical rules are both non-primitive, then failing to disambiguate them will result in the system's rejection of the rule the second time the user tries to introduce it, thus alerting him to his mistake. The way to resolve the ambiguity is to use different parts of speech for each language. The system, in this case, will interpret the rules as being different and accept both.

The only case in which the designer has to be aware of the need for separation is when one of the rules is a primitive. As described in chapter 6, defining two identical rules one of which is primitive and one non-primitive, in any order, results in accepting only the non-primitive rule. The failure to disambiguate the syntax will not be detected in this case.

The following example demonstrates what can happen in such a case. Suppose there are three languages: A very high level language called: High, the high level language: Algol and the target language: Macro Assembly. Suppose that both Algol and High use the same arithmetic expressions. The following Algol rules define two forms of expressions and translate them into an Assembly language stack-machine program:

(I) <EX> ::= <NU>
Subst: PUSH A,[<NU>]

(II) <EX> ::= <NU>$_1$+<NU>$_2$
Subst: MOVEI 1,<NU>$_1$
PUSH A,[<NU>$_2$]
ADDM 1,(A)

The word INCREMENT is used in High to increment expressions and is translated into Algol via the High - rule:

```
(III)  <EX> ::= INCREMENT <EX>
          Subst: <EX>+1
```

One form of a High - expression is introduced via the macro

```
MACRO
<EX>: 'N'
WHERE
N:<NU>
-->
PRIMITIVE
MEND
```

This macro has no effect on the dictionary since the primitive rule it attempts to introduce exists already in the dictionary as a non-primitive (rule I).

The High text segment: INCREMENT 5 parses successfully and yields the following parsing tree:

```
                     <EX> (III)
  |---------------------------------------------|
  |                                             |
                                            <EX>
                                             (I)
                                            |---|
                                            |   |

                                            <NU>
                                            |---|
                                            |   |
   I    N    C    R    E    M    E    N    T     5
```

In the semantic processing the arc <NU> returns the text 5. The arc <EX> (I) returns the text: PUSH A,5 and the latter is used by the semantic of the arc <EX> (III) to produce the text: PUSH A,5+1 which is not the desired target language output but a meaningless combination of Algol and Assembly language.

The thing that happened here is that in the translation of the text: 5 the translator "slipped" all the way down into Assembly language. 5 is an expression both in High and in Algol. The 5 in the input text is the High - 5, it had to be translated first into Algol and the resulting Algol text would translate correctly into Assembly language.

As already mentioned, the situation can be remedied by using a different part of speech for High expressions and for Algol expressions. If <HEX> is used in High and <EX> in Algol then rule III has to be replaced by the rule:

(IV) <HEX> ::= INCREMENT <HEX>
        Subst: <HEX>+1

and the primitive rule that was overridden by the non-primitive rule I should be replaced by the rule:

(V) <HEX> ::= <NU>
        Primitive

Now the text INCREMENT 5 yields the following parsing tree:

```
                     <HEX> (IV)
   |--------------------------------------------|
   |                                            |

                                           <HEX>
                                            (V)
                                           |---|
                                           |   |

                                           <NU>
                                           |---|
                                           |   |
     I   N   C   R   E   M   E   N   T       5
```

Now the semantic processing returns the Algol text: 5+1, which in the next translation iteration parses according to rule II whose semantic returns the correct target language text:

```
MOVEI 1,5
PUSH  A,1
ADDM  1,(A)
```

Note that in case two languages have identical primitive syntax rules then they can use the same parts of speech. An example is the rule:

<NU> ::= 5

which is both a High primitive and an Algol primitive. Actually if two languages have identical rules with identical semantics, then keeping only one rule in the dictionary is enough. Primitive rules are the most common special case of this category.

9 PIECEWISE TRASNSLATION

9.1 THE PROBLEM OF PROGRAM EXPLOSION

A typical procedure in this development system is a short program, with only a few lines of very high level code stating in general what the procedure does. Parts of the procedure are then translated via macros into a language in which the program is more refined. Parts of the new program are further translated, and so on until the whole program is defined in terms of the target language.

When a piece of text is refined, it is rewritten so that more details are revealed of the process, the data structure or whatever it describes. A more detailed description usually takes more text than the initial description, therefore refinements are mostly associated with expansions of the program. The amount of expansion in each step of refinement (ie - into how many new lines of code is one old line translated) depends on the style of the designer. The number of refinement steps from the initial program to the target language program also depends, for a given program and a given target language, on the designer: The more detailed the initial procedure is and the more is refined in each step - the less steps are necessary. Other factors that affect these numbers are the complexity of the task the program has to perform and the level of the target language.

Here are a few numbers from an actual program: In the first example in appendix C the expansion ratio of the macros is between 1 and 6 output lines (not counting BEGIN's and END's) for one input line, with an average larger than 3. The initial procedure consists of one line. The number of refinement steps for different parts of the program is between 2 and 5, and the target language (Simula) code is about 20 lines long. The second example in appendix C is an expansion of this Simula code into Macro assembly language. Here most of the expansions are in one step. The length of the output is about 200 lines whose length, on the average, is about 1/3 of the average Simula line length.

A fair estimate would be that the process of refining increases the amount of text at an exponential rate of about one order of magnitude for every 2 to 3 steps.

The phenomenon just described has practical implications on the translation process. Translation of an input text, as described in chapter 5, consists of several iterations. Each iteration constitutes one step in the process of refinement and therefore usually ends up in further expansion of the text. In every iteration the text has to be parsed and then processed semantically. The parsing algorithm is, on the average, worse than linear both in time and in space, and it turns out that, at the above mentioned rate of expansion, the simple iterative method becomes impractical after 4 to 6 steps. The only way to keep the translation process going and terminating after a reasonable time is to divide the text into segments which can be handled by the parser at a reasonable cost, and to translate each segment separately.

9.2 A PIECEWISE TRANSLATION METHOD

The question arises how can a program be split without affecting its successful translation. Clearly one can not split it arbitrarily since this will, in general, result in segments which do not parse and hence do not translate separately. The segments into which a program is split have to be logical units that are independent of one another in their translation.

In order for a text segment to be translatable it has to parse, ie - there has to be an arc which spans the whole segment. It is the semantic of this arc which determines the translation. This criterion can be used for splitting a long text. If the parsing tree is given, subtexts which are covered by arcs can be translated without referring to the rest of the text. Using the same criterion again these subtexts can be split into shorter segments and so on until the leaves of the tree are reached.

Using this method, the parsing tree on page 46 can be processed in the following way: The subtext SUM OF 'A' AND 'B' is covered by an arc, thus it can be processed separately. The sub-subtexts 'A' and 'B' are each covered by an arc, so they can be processed separately. Both of them return the original texts. The processing of these texts is now completed, they are marked as translated and their parts of speech are recorded for future use. Now SUM OF 'A' AND 'B' is translated using the semantic of the arc <EX> (IV) which returns 'A'+'B'. Since the arc was not the result of a primitive, the text undergoes another translation iteration. The first step in this iteration is replacing the subtexts 'A' and 'B' , which are marked as translated, by single arcs with parts of speech <ID> whose semantics return the original texts. Note that initially each character is represented by an initial arc, thus each replacement reduced three arcs to one arc in this case). The string that is passed to the parser is:

```
<ID>        <ID>
|---|       |---|
|   |   +   |   |
```

It parses to <EX> by rule V (on page 45). This rule is a primitive and the text remains unchanged. It is marked as translated, and its part of speech <EX> is recorded.

Another subtext that is covered by an arc is the parameter 'B' (following the word SET). It is translated separately, marked, and its part of speech <ID> is recorded.

Now comes the turn of the arc <STMT> (II). Its semantic returns the text: 'B':='A'+'B' in which the subtexts ''B' and 'A'+'B' are marked as translated. In the second iteration the marked texts are replaced by single arcs. The input to the parser is the string:

```
<ID>              <EX>
|---|             |---|
|   |   :    =    |   |
```

It parses by rule III (page 45) which is a primitive. The text remains unchanged, and the translation process is completed.

9.3 STATE TRANSITIONS AS BASIC TRANSLATION UNITS

The method described in the previous paragraph reduces the amount of parsing to a minimum, but it has a drawback and therefore is only partially used. The problem with the method is that it takes subtexts out of their contexts. As mentioned in chapter 8, two or more languages may share syntax rules. Certain text segments may parse ambiguously by the rules of more than one language, and it is the context in which the text appears that determines which of the parsings is relevant and which is spurious. But, as the example shows, in the translation process subtexts are parsed separately and if a subtext, like 'A'+'B' in the example, belongs to two languages, there is no way to tell which parse to pick.

Subtexts can also be taken out of their context within one language. Here is an example of such a case: Let the languages be, like in the example of chapter 8, High, Algol and Macro assembly, with the following non-primitive rules:

Algol rules:

(I) ⟨EX⟩ ::= ⟨NU⟩
Subst: PUSH A,[⟨NU⟩]

(II) ⟨EX⟩ ::= ⟨NU⟩$_1$+⟨NU⟩$_2$
Subst: MOVEI 1,⟨NU⟩$_1$
PUSH A,[⟨NU⟩$_2$]
ADDM 1,(A)

High rules:

(III) ⟨HEX⟩ ::= FIVE
Subst: 5

(IV) ⟨HEX⟩ ::= INCREMENT ⟨HEX⟩
Subst: ⟨HEX⟩+1

Note that in this case no texts are shared by the languages. Suppose that 5 parses to ⟨NU⟩. Let the input be: INCREMENT FIVE . The parsing tree is:

```
                               <HEX> (IV)
|--------------------------------------------------------|
|                                                        |
                                         <HEX> (III)
                                    |---------------|
                                    |               |
  I   N   C   R   E   M   E   N   T    F   I   V   E
```

The subtext FIVE is covered by an arc and translated into: 5 . The resulting text undergoes a second iteration. The parsing tree is:

```
  <EX>
  (I)
 |---|
 |   |

  <NU>
 |---|
 |   |
   5
```

And the translation, according to the semantic of rule (I), is: PUSH A,5 . Now the semantic of the arc <HEX> (IV) is applied, yielding the text: PUSH A,5+1 which is meaningless, does not parse, and the translation ends with the wrong result.

The reason for obtaining the wrong text is that the text 5 was taken out of its context in the second iteration. Had the whole text been translated and then reiterated, then the input to the parser in the second iteration would have been the text: 5+1 . There is only one way to parse this text, namely by rule (II), and the arc <EX> (I), that the parser would build around the subtext 5, would be spurious and not participate in the parsing tree.

The above described problem makes it clear that not every subtext of a program, even if it is meaningful, can be translated separately. Still, for a system to be practical, it is necessary to split the program. So the question arises how can subtexts whose translations do not depend on the environment be identified?

There is no general solution to this question. It all depends on the syntax of the languages the designer uses. However, keeping in mind that the designer writes a computer program and does not just play games with syntax rules, there is a class of subtexts which are independent of each other. To see what this class is, one has to look at the real semantics of the language constructs used in the program. By <u>real</u> semantics I mean the operations that the constructs refer to as oppposed to the translation system semantics which are texts in another language into which the constructs have to be translated. One way to look at these semantics is described by Dijkstra in [9]. Dijkstra looks at the set of states defined by the variables of the program and views the program as a transition (in his terminology: A predicate transformer) between specified initial and final subsets of this set. In most, even least complex programs this transition is composed of a series of smaller state transitions. Each of the intermediate transitions has its own sets of initial and final states. These sets are independent of the other state transitions, which can at most limit the set of initial states the transition may actually encounter in a particular program to a subset of those on which it works.

The translation of a text segment whose real semantics are a state transition in language L results in a text in another language L1. L1 is at least as expressive as L since the set of states it describes has to include L's set of states. The real semantics of the text in L1 describe the same state transition as the L-text but it may be composed of a series of finer transitions between states that L can not distinguish. A typical example is the translation of an Algol text, say, U:=V+W into assembly code. The set of states, which the Algol speaker can see, consists of all the combinations of values assigned to the variables U, V and W. The set of states of Assembly language includes all those, but the assembly language speaker also talks about registers, stacks, addresses etc and breaks the single Algol transition into two or three sub-transitions expressed in those terms.

A given state transition is something that occurs in the real objective world. It remains the same transition independent of the language that describes it. Therefore translation of a text that describes a state transition into another language can not be influenced by other texts that might be in the neighbourhood. The answer to the piecewise translation problem follows from this fact. If a program is split into subtexts which correspond to real world state transitions, then each subtext can be translated separately and the translation would be correct.

The problem remaining now is how can these subtexts be recognized. The translation system does not know anything about the real semantics of the texts it translates; it has to be told by the user what text segments correspond to state transitions. A natural way to indicate these texts is via parts of speech. A look at syntax descriptions of programming languages shows that most of them use the word "statement" for texts that correspond to state transitions. This convention was adopted here too in order to make things as natural as possible. Every part of speech whose last four characters are STMT (in particular the part of speech <STMT> itself) is viewed by the translator as referring to a text which is separately translatable, and <u>only</u> these texts are translated separately.

The choice of parts of speech is done at the user's wild imagination discretion. There is no way for the translator to check whether texts that parse to <...STMT> really are separately translatable. However it does not seem to be to much of a strain for the designer, it really looks quite natural, to assign a part of speech ending with STMT to statements. In order to encourage this way of design, <STMT> is the default lhs part of speech of macros and procedures. For example, macro 10 on page 35 could have been written:

```
MACRO
SET 'U' TO 'V'
WHERE
A,B: <ID>
-->
'U':='V'
MEND
```

A text does not necessarily have to be a state transition description to start with, in order to be split into state transition descriptions at a later refinement. For example, the expression: A+B is not a state transition in the Algol level. If it is refined into assembly language, it is translated into code that leaves its value in some location - the accumulator, for example - so it may be translated into the two instructions:

```
LOAD A
ADD  B
```

Each of these instructions constitutes a state transition on the assembly language level.

Note that a text can only be split after it has been parsed. An input text, regardless of the number of statements it includes, has to be parsed as a whole in order to obtain the parsing tree from which the statements can be separated. Only from the second translation iteration on are statements translated separately. For this reason it might be of advantage to write procedures in a very high level, general form and refine them with macros rather than to start with more refined forms. Suppose, for example, that a procedure's initial text consists of the statements S T , that S translates into S1 S2, and T - into T1 T2 . The text that enters the second translation iteration is: S1 S2 T1 T2 . Since the input text S T was parsed, the system knows that the translations S1 S2 and T1 T2 are statements and should be considered separately for further refinements. If the designer wrote the procedure in the more refined form in the first place, then the initial text would be S1 S2 T1 T2, and it would parse as a whole and not be separated until the next iteration. All it takes to achieve an early separation of texts is using macros in order to start the design at a level that is as high as possible. It contributes to the clarity of the design and, on the other hand, does not affect the final program since the macros disappear in the translation process.

10 USING SDS FOR OTHER TASKS

10.1 FIXING THE TARGET LANGUAGE, THE POL SYSTEM

Being a general development system, one of its objectives is the ability to use any desired language as target language. However, as discussed in 4.1, in most cases users tend to choose a language out of a small set for their programs, so that one may write a large number of designs, all with the same target language. In such a case a natural extension of the software development system is to augment it with the target language compiler and produce executable machine code directly from the design.

A project along these lines is the POL (Problem Oriented Language) system developed by Dr. Fred Thompson at Caltech. POL enables the user both to build and to use application languages. The tools of the software development system together with a compiler for Pascal, which is the fixed target language, are incorporated into POL as parts of its metalanguage - the language in which the object (application) language is written.

POL has two dictionaries: An object language dictionary and a metalanguage dictionary (which corresponds to the user's dictionary in the development system). An object language is created at the metalanguage level by defining object language syntax rules (in short: rules), macros, procedures and parts of speech.

Rules, like procedures and macros, have a left-hand side and a right-hand side. The left-hand side is a syntax rule of the object language, which is merged into the object language dictionary. The right-hand side is the corresponding semantic routine, which may be written in any language of the writer's choice. The routine is first translated into Pascal - using the metalanguage dictionary, and then compiled. The resulting machine code is stored, and its location is linked to the syntax rule in the object language dictionary.

Macros provide the translation rules which are inserted into the metalanguage dictionary. Of course, the metalanguage will only work successfully if every non-Pascal structure is translated all the way down into Pascal via appropriate macros. All the Pascal primitives are pre-merged into the metalanguage dictionary, so that the designer can freely mix very high level structures with Pascal code.

It is the handling of procedures where the facts that the target language is fixed and its compiler is incorporated into the system, make the difference. Recall that in SDS the left-hand side rule of a procedure serves as its calling sequence. The formal parameters in this rule stand for non-terminal parts of speech and may be replaced in the procedure call by any text with the appropriate structure, ie - a text that parses to one of the parts of speech which correspond to the particular parameter. Pascal procedures, rather than being called by an arbitrary structure as SDS procedures are, are called by single names followed by an argument list. Like SDS procedures Pascal procedures also accept arguments only if they are of the correct structure.

But rather than calling these structures "parts of speech", they are called "data types" and the syntax for their declaration is different from the SDS syntax. The above mentioned part of speech declarations, which are part of POL's metalanguage, are used to associate parts of speech with Pascal data types. Only after having appeared in such a declaration may a part of speech be used in a procedure.

In view of all these facts procedure definitions are handled in the following way: The left-hand side rule is inserted into the metalanguage dictionary, but this time not as a primitive. An internal name is assigned by the system to each procedure; this name is put in the dictionary as the semantic part of the left-hand side rule, ie - whenever the translator encounters the left-hand side rule (which, for the writer, serves as the procedure's calling sequence) it will be translated into the internal name (which serves as the Pascal calling sequence). The right-hand side text is first translated into Pascal. The resulting text is prefixed with a Pascal procedure declaration header including the internal name and the data types corresponding to the arguments' parts of speech; all this is compiled, and the machine code is stored and linked with the procedure's internal name.

Procedures may be used as semantic routines of rules, they may also be called by semantic routines or by other procedures.

One language processor is used for handling both the object language and the target language. Command-language commands enable the language writer to switch between the two languages so that he can easily iterate between writing the object language and using it (or trying it out).

10.2 PORTABILITY AND ADAPTABILITY OF SDS PROGRAMS

One of the objectives of a software development system, as discussed in chapter 3, is to produce programs which are portable and adaptable. The main objectives towards which SDS was designed are the ones which concern the development itself, namely: Flexibility in design, free choice of the programming language and automated coding. However the design can be used to make changes in the programming language or in the program's specifications. How easy it is to make these changes - in other words: how portable and adaptable is the program - is the subject of the following discussion.

No matter how "unorderly" the act of design was - top-down, bottom-up or any mixture of them - it always results in a series of languages, for each procedure, that constitutes a hierarchy ordered in increasing degrees of expressiveness, through which the translator proceeds in translating the procedure. If a program is to be rewritten in a new language L1 whose set of states is a subset of a language L from this hierarchy, then, since every state transition of L can be expressed in terms of state transitions of L1, the program can be written in the new language by replacing the macros translating L into the old target language with macros translating L into L1. Of course every language which stands above L in the hierarchy, and therefore is at most as expressive as L, can be used instead of L as a starting point for the change.

All this sounds very simple in theory. In practice however the ease of writing the set of new macros translating L into L1, or - the portability of the program, depend on the relationship between the two languages. The simplest case, which rarely happens, is when the new language L1 in which the program has to be written is in the hierarchy. Then L is taken to be identical to L1, and one only has to delete from the design all the macros which translate it further down thus making it the new target language.

Another simple case is when a programming language has to be replaced by a different dialect of itself. This happens, for example, if the compiler has been replaced or if a program has to be transferred to another computer which supports the same language. In such cases usually some language constructs have to be modified to fit into the new environment. L1 is in this case the new dialect and, if L is taken to be the old dialect, a rather small set of macros can provide the necessary rules for translating it into the modified version.

More effort is required if the programming language has to be replaced by one of much lower level. If L is taken to be the old programming language a suitable set of macros can be written that links the lower level language to the bottom of the hierarchy. An example of this case is shown in appendix C where Simula code is translated into Macro Assembly language.

In both cases just described it was possible to take the old programming language as a starting level for the creation of new levels. The macros that perform these translations are program independent and can serve as a translation package to transfer all the programs from the old programming language into the new one.

The task becomes more difficult if the two languages are of more or less similar levels but are different in nature. Suppose, for example, that a Pascal program has to be rewritten in LISP. A set of macros that express Pascal constructs and data types in terms of LISP lists can be written, but it is hard to believe that a programmer would choose this way, because a great effort is required to write the macros that perform this kind of translation and the resulting LISP program will be longer, less efficient and more difficult to understand and to maintain than a program that was designed for LISP in the first place. The difficulty arises from the fact that each of these languages requires a different programmig style and a different approach to the task. It would be much easier in such a case to start the translation from a level in the language hierarchy where the influence of the target language is not yet felt. This level might very likely be the highest one which corresponds to redoing the design from the beginning. From the language theoretic point of view the situation can be described as L and L1 having similar, but not identical, sets of states. The language up in the hierarchy which serves as a convenient starting point is one whose set of states is a subset of the sets of both L and L1.

As the above discussion shows, portability does not always depend on the user's style. If the transition to the new programming language can be made with the old language as a starting level, then the design does not affect the complexity of this task. On the other hand, if the starting point is a language introduced by the designer himself, then of course the ease of moving to a different target language depends on what has been designed so far.

Adaptability - the ease of changing the program in order to meet new specifications - on the other hand, depends, for a given change, entirely on the design of the program. If the design is clean in the sense that interaction - and hence: dependence - between parts is kept to a minimum, then the change should be relatively easy to make as it does not affect many parts. Otherwise it might cause a chain reaction of necessary changes in a large number of procedures and macros. Sds does not police the amount of dependence between modules of the program. It provides tools which, if properly used, can yield adaptable programs.

11 SUMMARY

11.1 OVERALL OUTLINE

The overall goal of the work described in this thesis was to build a system that supports the activity of design and coding of a software system by requiring from the user only the minimum that will enable the system to take over the rest of the task.

This goal led to the following objectives such a system should meet:

(i) Flexibility of design. This objective has two aspects:

- Ability to start the design at any point and proceed in any direction: Top-down, bottom-up or any combination of them.
- No prescribed constructs and no restrictions to the language used for the design.

(ii) Automated production of code by the system once the design is completed.

(iii) Ability to use any language as target language and no need for special specification of this language other than its appearance at the bottom level of the design.

(iv) Production of programs that are portable and adaptable.

To meet objectives (i), (ii) and (iii) SDS was build as a two-pass system whose second pass is performed by a language processor which uses a powerful parser. The user writes his design modules in this system either as macros or as procedures. Both macros and procedures define a refinements - or translations - of constructs from one language into constructs of another language, and at the same time serves as a means of introducing the appropriate syntax rules into a dictionary. To turn the design into a program SDS translates the procedures according to the tanslation rules defined in the macros. Thus the procedures become the modules of the final system while the macros disappear.

The ability to do the design in any direction is achieved by the use of two passes: In the first pass the syntax rules of the designer's languages and their semantics are introduced into the dictionary. In the second pass the translation is perofrmed according to these rules.

The use of any language is made possible by the use of the powerful parser.

Automated coding into the bottom level language is done by the semantic routines of the language processor which iterates by translating the procedure bodies into lower and lower languages according to the translation rules stored in the dictionary and stops when the output text is constructed by primitive rules only.

The system was designed mainly to meet objectives (i), (ii) and (iii). However the hierarchy of languages which the user creates can provide a convenient staritng point for rewriting a program in a different language.

11.2 SPECIFIC PROBLEMS

The following problems had to be solved in order to make the system work correctly and satisfactorily:

11.2.1 HANDLING OF AMBIGUITIES

The SDS translator may encounter several kinds of ambiguities. In order to obtain a correct translation each kind has to be recognized and treated in a different way. Here is a short description of each kind of ambiguity and of the way it is handled in the system:

(i) Ambiguities which cause error messages. If the parsing graph of a text to be translated has two or more spanning arcs with different parts of speech, the system can not resolve the ambiguity. In such a case an error message is issued and the translation is aborted. All other ambiguity cases discussed below have two or more spanning arcs with identical parts of speech.

(ii) Internal ambiguities introduced by the user. The user can freely use ambiguous rules a long as the ambiguity is only internal and can be resolved by the context in which the ambiguous construct occurs. This kind of ambiguity is handled by the parser which does not use the arcs which do not fit into the context and thus makes them spurious.

(iii) Ambiguities due to special case definitions. The user may define a translation rule which specifies that a subset of a set of text-strings is to be handled in a different way than the rest of the set. The parsing of a text which includes one or more special cases results in several parsing trees. The translator assures that all special cases are included in the translation by picking the tree with the minimum number of branches.

(iv) Ambiguities which cause warning messages. If more than one tree with the minimal number of branches are found, the reason might be either the occurence of different special cases or a user's error or both. In such a case the translator arbitrarily picks one of the trees and issues a warning to alert the user to the possibility of an error.

(v) Ambiguities due to indirectly redefined implied primitives. This kind of ambiguity occurs if the user redefines an implied primitive in parts rather than as a whole. In such a case the implied primitive is not automatically removed from the dictionary and its participation in parsing leads to ambiguous parsing graphs. The tanslator resolves this ambiguity by picking only the parsing trees with the minimun number of implied primitives as candidates before counting the branches.

11.2.2 LANGUAGE SEPARATION

SDS uses one dictionary to store all the syntax rules defined by the user regardless of the language to which they belong. In this way the user enjoys the maximum amount of flexibility in the use of these languages. He can move in any direction, mix languages and skip levels of the language hierarchy in his translation rules without having to do any bookkeeping of the hierarchy which he defines.

A result of this fact is that two or more languages may not share identical syntax rules which have different translations. The way to distinguish between two identical sonstructs which belong to different languages is to use different parts of speech. If one rule is primitive and the other non-primitive, then the user's failure to use different parts of speech will remain undetected due to the lower priority of primitive rules. In the other cases, when both rules are non-primitive, a user's error will result in the rejection of the second rule and an error message will be issued.

11.2.3 HANDLING LONG TEXTS

In each translation step of SDS texts are, in general, refined and become longer. Every 2 to 3 steps may increase the length of the text by an order of magnitude.

SDS uses a powerful parser which is able of handling any text, but consumes large amounts of time and storage space if texts become long.

Handling each text which syntactically is separately translateable may result in translation errors due to the fact that in this way texts are taken out of their contexts.

The SDS solution is to handle separately only texts which describe state transitions, whose translation does not depend on their environment. The translator recognizes such texts with the help of the user. Every text which parses to a part of speech whose last four characters are STMT is treated as separately translateable. Since one usually writes and refines statements, and since <STMT> is the system's default part of speech, this way of guiding the translator is quite natural and should not require extra effort on the user's part.

11.3 PRESENT STATUS

SDS has been implemented on the DEC-20 computer in Simula. A second system is currently being implemented in Pascal as part of the metalanguage of the POL (problem oriented language) a system for writing and using application languages. SDS will be used in POL for designing semantic routines of syntax rules of new - or extensions of old - languages.

Appendix A

USER'S REFERENCE MANUAL

A.1 COMMAND LANGUAGE

SDS runs on the DEC-20 computer and uses the TOPS-20 file structure the knowledge of which is assumed in this manual. When the system is entered, it is in command language level. The prompt: > indicates that a command may be issued. Here is a list of the commands and their effects in alphabetical order:

A.1.1 CLEAR

Erase the user's dictionary. The system responds with CONFIRM: and waits for YES in which case the dictionary is erased, or NO in which case nothing happens.

A.1.2 DEBUG

Puts the translator in debug-mode where the translation is interrupted in each iteration so that the user can look at useful data for debugging the syntax of his design (see paragraph A.3).

A.1.3 DICTIONARY

Type the contents of the user's dictionary on the terminal. Every rule is typed with an arrow (-->) separating the right-hand side from the left-hand side. If two or more parts of speech stand for identical strings (see paragraph A.2.4) then the following line contains a list of their numbers separated with equal symbols (=). This is followed by the semantics. If the rule is non-primitive the semantics consist of the word SUBST: followed by a string of numbers between single quotes and of characters. The numbers indicate parts of speech of the syntax rule counting from right to left. The count includes terminal as well as non-terminal parts of speech. Strings of blanks are counted as one part of speech. If the rule is a primitive or an implied primitive, then the semantics line consists of this information. Following the semantics line is the source - or record - file name without extension and the line number in the REF file (see paragraph A.2) of the macro or procedure in which the rule was defined.

If, for example, the source - or record - file name is EXAMPLE, and the input consists of the three macros:

```
MACRO
<STMT>: (SET 'Y' 'E')
WHERE
Y:<ID>  E:<EX>
-->
'Y':='E'
MEND
```

```
MACRO
<EX>: 'E1'+'E2'
WHERE
E1,E2:<EX>
-->
PRIMITIVE
MEND


MACRO
<EX>: 'E'*'E'
WHERE
E:<ID>,<NU>
-->
'E'**2
MEND
```

Then the DICTIONARY command will display the following text on the terminal:

```
<STMT>--> (SET <ID> <EX>)
Subst: '4':='2'
     Def: EXAMPLE -   1

<STMT>--> <ID>:=<EX>
     Implied Primitive
     Def: EXAMPLE -   1
```

```
<EX>--> <ID>*<ID>
       1=  3
Subst: '1'**2
     Def: EXAMPLE -  19

<EX>--> <ID>**2
     Implied Primitive
     Def: EXAMPLE -  19

<EX>--> <EX>+<EX>
     Primitive
     Def: EXAMPLE -  10

<EX>--> <NU>*<NU>
       1=  3
Subst: '1'**2
     Def: EXAMPLE -  19

<EX>--> <NU>**2
     Implied Primitive
     DEFINITION No:  19
```

A.1.4 ENDDEBUG (or ENDEBUG)

Terminate debug-mode.

A.1.5 ENTER

Load a dictionary from an external file. The system responds with FILE NAME: After the file name has been entered the new dictionary is loaded and the dictionary that existed prior to issuing the command is erased (see also the SAVE command).

A.1.6 EXIT

Terminate and exit to monitor level.

A.1.7 LBRACKETS

Change the brackets used to identify labels in macros. The user is prompted for each bracket. Hitting the return-key leaves it unchanged.

A.1.8 PBRACKETS

Change the brackets which identify parameters in macros. The user is prompted for each bracket. Hitting the return-key leaves it unchanged.

A.1.9 PRIMITIVES

Same as DICTIONARY but only primitive and implied primitive rules are output.

A.1.10 SAVE

Save the user's dictionary in an external file for future use. The system responds with: FILE NAME (EXTENSION:

'SAV'): and waits for the user to enter a file name. The extension of the file name is changed to SAV if not so already.

A.1.11 SYNTAX

Build the user's dictionary. For details see paragraph A.2.

A.1.12 TRANSLATE

Translate procedure bodies. The system responds with a request for an input file name. If no other TRANSLATE command has been issued since the last SYNTAX command, then hitting the RETURN key indicates that the default file should be taken. The default file is the input - or record - file used in the last SYNTAX command.

Two output files are created. One of them contains the translated procedures in the target language. Its default name is the input file name with extension changed to TGT, but it may be overridden by the user. The second file contains a list of the macros used in the translation of each procedure in their order of use. Its name is the input file name with extension changed to TRS. Both files are considered as old files, ie - if a file with the identical name already exists in the user's directory, the new text is appended to it.

A.1.13 TYPE

Type a file on the terminal. The system responds with a request for the file name.

A.1.14 UPDATE

Remove all implied primitive rules which have been re-defined indirectly from the dictionary (see paragraph 7.4).

A.2 THE SYNTAX PASS

A.2.1 FILE HANDLING

All the user's input is done in the syntax pass. Macros and procedures may be input either from an external file or interactively from the terminal. When the command SYNTAX is issued, the system requests an input file name. The response TERMINAL indicates that definitions will be input interactively. Any other response is interpreted as an external file name from which the definitions should be taken.

If the input mode is interactive, the system requests a name for a record file. All the user's input will be recorded in this file in a format that matches the syntax that has to be used in an external input file, so that the record file may be used at another time as an external input file in order to enter the same definitions. The record file is considered old - this means, that if a file with the same name exists already in the user's directory, then the new input is appended to its end.

In both input modes - external and interactive - a reference file is created. Its name is the input - or record - file name with extension changed to REF. The reference file contains the same text as the input - or record - file with the addition of line numbers and, if the input is external, error messages if any (if the input is interactive, the user is requested to re-type erroneous input lines until they are correct, only then is the input recorded in the record and reference files). The reference file is considered old if the input is interactive, or new - any previous contents of the file are erased - if the input is external.

The file handling facilities described above and in paragraph A.1.10 and the possibility to save and restore the user's dictionary are intended to support design and production of large programs by single users or by groups. It takes many sessions to create a large program. The SAVE and ENTER commands enable the designer to save the dictionary he has built in an external file at the end of a session and to restore it at the next one without having to waste time on rebuilding it. The choice of appending information to old output files or creating new ones in interactive syntax mode and in the translation pass provides the flexibility needed for organizing the design documents and the source code in any desired files structure.

A.2.2 EXTERNAL INPUT

The following syntax should be used for entering definitions via an external file. Square brackets ([]) indicate options, lower case letters are used for descriptions and should not be taken literally:

Macro definitions

```
<MACRO>       ::=  <LHS>
                   -->
                   <RHS>
```

Left-hand side

```
<LHS>         ::=  [<POS> <COLON>] <LHS TEXT>
                   [WHERE
                    <DECLINES>]
```

If the part of speech is omitted, then <STMT> is taken as default. If the left-hand side text contains no parameters, then the declarations have to be omitted. Undeclared parameters are considered declared as <ID> by default.

Part of speech

```
<POS>         ::=  <LPOSBRKT> <ID> <RPOSBRKT>
```

Part of speech brackets

```
<LPOSBRKT>   ::=  <
<LPOSBRKT>   ::=  < <BLANK>
<RPOSBRKT>   ::=  >
<RPOSBRKT>   ::=  <BLANK> >
```

Blank

```
<BLANK>      ::=  blank
<BLANK>      ::=  <BLANK> blank
```

Colon

```
<COLON>      ::=  :
<COLON>      ::=  <BLANK> :
<COLON>      ::=  : <BLANK>
<COLON>      ::=  <BLANK> : <BLANK>
```

Left-hand side text

<LHS TEXT> ::= <STRING>

<LHS TEXT> ::= <PARAMETER>

<LHS TEXT> ::= <LHS TEXT> <STRING>

<LHS TEXT> ::= <LHS TEXT> <PARAMETER>

String

<STRING> ::= string of characters not including parameter and label brackets

Parameter

<PARAMETER> ::= <LPARBRKT> <ID> <RPARBRKT>

Parameter brackets

<LPARBRKT> ::= initially a single quote ('); may be overridden with any character other than letters, label brackets and part of speech brackets; optionally followed with blanks

<RPARBRKT> ::= initially a single quote ('); may be overridden with any character other than letters, label brackets and part of speech brackets; optionally preceded with blanks

Declaration lines

<DECLINES> ::= <DECL>

<DECLINES> ::= <DECLINES> <DECL>

Declaration list

<DECL> ::= <DEC>

<DECL> ::= <DECL> <BLANK> <DEC>

Declaration

<DEC> ::= <IDL> <COLON> <POSL>

Identifier list

<IDL> ::= <ID>

```
<IDL>          ::=  <IDL> <COMMA> <ID>
```

Identifier

```
<ID>           ::=  <LETTER>

<ID>           ::=  <ID> <LETTER>

<ID>           ::=  <ID> <DIGIT>

<ID>           ::=  <ID> _
```

Comma

```
<COMMA>        ::=  ,

<COMMA>        ::=  <BLANK> ,

<COMMA>        ::=  , <BLANK>

<COMMA>        ::=  <BLANK> , <BLANK>
```

Part of speech list

```
<POSL>         ::=  <POS>

<POSL>         ::=  <POSL> <COMMA> <POSL>
```

Right-hand side

```
<RHS>          ::=  [[<POS> <COLON>] <RHS TEXT>]
                    MEND
```

If the part of speech and colon are missing, the left-hand side part of speech is default.

Right-hand side text

```
<RHS TEXT>   ::=  <STRING>

<RHS TEXT>   ::=  <PAR>

<RHS TEXT>   ::=  <LABEL>

<RHS TEXT>   ::=  <RHS TEXT> <STRING>

<RHS TEXT>   ::=  <RHS TEXT> <PAR>

<RHS TEXT>   ::=  <RHS TEXT> <LABEL>
```

Label

```
<LABEL>        ::=  <LLABBRKT> <LID> <RLABBRKT>
```

When a label is encountered in the translation process, its brackets are removed, and it is augmented with an integer which is unique to the application of the macro in which the label appears, so that in another application of any macro in which this label may occur it will be augmented with a different number.

Label identifier

<LID> ::= <ID>

<LID> ::= <NU>

Number

<NU> ::= <DIGIT>

<NU> ::= <NU> <DIGIT>

Digit

<DIGIT> ::= any digit

Label brackets

<LLABBRKT> ::= initially a dollar symbol ($); may be overridden with any character other than letters, parameter brackets and part of speech brackets; optionally followed with blanks

<RLABBRKT>	::=	initially a dollar symbol ($); may be overridden with any character other than letters, parameter brackets and part of speech brackets; optionally preceded with blanks

Procedures

```
<PROC>         ::=  PROC
                    <LHS>
                    -->
                    <LHS TEXT>
                    PEND
```

Macros and procedures may be preceded with the word DELETE. This causes the syntax rules to be deleted from the dictionary, if they are found there, rather than inserted. DELETE applies only to the macro or procedure immediately following it.

Comments are lines starting with an exclamation mark (!) the may be inserted between definitions. They are copied into the REF file and are other wise ignored.

A.2.3 INTERACTIVE INPUT

If the input mode is interactive, then, after obtaining a record file name, the system outputs a double prompt: >> to indicate that it is ready for input. Now the user can enter one of the words MACRO, PROC, DELETE, EXIT or a comment.

The DELETE command indicates that the syntax rules defined in the immediately following macro or procedure should be deleted from the dictionary. The system responds to the command with a double prompt. The command applies only to one macro or procedure definition. After the definition has been entered, the user is reminded of this fact by the text: DELETE MODE CANCELLED.

The EXIT command puts the system back into command language level.

A comment is a line starting with an exclamation mark (!). It is copied into the record and reference files and is otherwise ignored.

The MACRO and PROC commands are used to enter macro and procedure definitions respectively. They are responded with series of input requests for the various parts of the definition. The syntax for entering these parts is identical to the syntax used in an external input file. The termination of multiple line entries (left-hand side and right-hand side texts and declarations) is indicated by a line consisting of a percent sign (%). Each declaration line is processed separately and the system issues a triple prompt: >>> to indicate that the next declaration line may be entered.

Here is an example of interactive input. Texts input by the user are indicated with underscores.

```
>>MACRO

LHS PART OF SPEECH: <EX>

LHS TEXT:

F('X', 'Y')
%

DECLARATIONS

>>> X,Y: <NU>,<ID>

>>>%

RHS PART OF SPEECH:

RHS-POS IS: <EX>

RHS TEXT:

'X'*('Y'-8)
%

>>
```

A.2.4 SPECIFYING IDENTICAL TEXTS

Identical parameters in the left-hand side of a macro or a procedure or in the right-hand side of a macro stand for identical texts (and hence - identical parts of speech).

This is recorded in the dictionary, and texts will parse to the corresponding rules only if the texts in the appropriate places really are identical.

For example consider the third macro of paragraph A.1.3 . The parameter E occurs twice in the lhs text and it represents two parts of speech, but only two lhs rules (rather than four) are inserted into the dictionary, and the fact that both parts of speech have to represent identical texts is noted following the rule when the DICTIONARY command is issued.

A.2.5 COMMENTS

In both input modes, every line whose first non-blank character is an exclamation mark (!) is considered a comment. Comments entered in interactive input are transferred into the record file.

A.3 DEBUG MODE

Debug mode is intended to aid the user in detecting errors in his own syntax and in its use. If a procedure does not translate correctly, the reason usually is that it, or sections of it, did not parse, either because of an error in a syntax rule in the user's dictionary, or because of an error in the procedure text. In debug mode the user can follow the translation step by step in order to detect his errors.

The command DEBUG is responded with a request for a file name. All the related outputs will be put in this file. If the name entered is TERMINAL then the outputs will appear on the user's terminal rather than in an external file.

In debug mode the translator stops its processing every time when parsing is completed and issues a double prompt: >>. At this stage the user may type one of the commands: TEXT, PMARKER, GO or a part of speech.

The GO command instructs the translator to resume processing.

The TEXT command outputs the text being currently translated.

The PMARKER command outputs the whole parsing graph in a table.

Typing a part of speech results in outputting all the subtexts of the textstring currently being translated that parsed into that particular part of speech.

If, for example, the text being currently translated is: Y=F(A) with parsing graph:

```
           <STMT>
  ¦----------------------¦
  ¦                      ¦

     <STMT>
  ¦-----------¦
  ¦           ¦

                <EX>
          ¦--------------¦
          ¦              ¦

          <FN>
          ¦---¦
          ¦   ¦

  <EX>    <EX>    <EX>
  ¦---¦   ¦---¦   ¦---¦
  ¦   ¦   ¦   ¦   ¦   ¦

  <ID>    <ID>    <ID>
  ¦---¦   ¦---¦   ¦---¦
  ¦   ¦   ¦   ¦   ¦   ¦
    Y   =   F   (   A   )
```

and if the output is directed to the user's terminal, then the following conversation might take place:

```
>>TEXT

Y=F(A)

>>PMARKER

INITIAL ARC         ALTERNATE ARCS        NEXT(OF ALTERNATE)
============================================================
'Y'
                      <ID>                     '='
                      <EX>                     '='
                      <STMT>                   '('
                      <STMT>                   ---
------------------------------------------------------------
'='
------------------------------------------------------------
'F'
                      <ID>                     '('
                      <FN>                     '('
                      <EX>                     '('
                      <EX>                     ---
------------------------------------------------------------
'A'
                      <ID>                     ')'
                      <EX>                     ')'
------------------------------------------------------------
')'
------------------------------------------------------------
```

```
>><STMT>

Y=F
--------------
Y=F(A)
--------------

>><EX>

Y
--------------
F
--------------
F(A)
--------------
A
--------------

>><FN>

F
--------------

>>
```

A.4 CHARACTER HANDLING

A.4.1 LETTERS

All letters in commands, sub-commands and keywords may be typed in lower-case as well as in upper-case (or, for this matter, any combination of them).

Names are considered to refer to the same parameter or part of speech regardless of whether their letters are lower-case or upper-case.

Lower-case letters parse to the same part of speech as the corresponding upper-case letters. Thus rules which have been defined in, say, lower-case may be used in upper-case without affecting the success of parsing.

A.4.2 CONTINUATION CHARACTERS

A backslash (\) at the end of an input line indicates that the line is to be continued. If the backslash is immediately preceded with a dash (-), both characters and the following <CR><LF> are removed thus concatenating the following line to the character immediately preceding the dash. Otherwise the backslash and the following <CR><LF> are removed and a blank is inserted between the two lines.

For example, the input:

```
Cal-\
tech
```

is the same as:

```
Caltech
```

and the input:

```
Los\
Angeles
```

is the same as:

```
Los Angeles
```

A.4.3 BLANKS AND <CR><LF>

Strings of one or more blanks are handled like one blank both in definitions (syntax pass) and in procedure bodies (translation pass).

If a rule, introduced by the user in syntax pass, consists of more than one line, each <CR><LF> is replaced by the part of speech <BLANK>. However non-leading <CR><LF>'s of rhs texts are remembered in the semantic part so that the output from the translator is formatted in the way prescribed by the user.

In translation pass each <CR><LF> in the procedure body parses as <BLANK>. Thus a text written in one line can parse by a rule that was enterd in several lines and vice versa.

Appendix B. IMPLEMENTATION

B.1 THE SYSTEM'S STRUCTURE

The software development system is written in Simula on the Dec-20 computer. The system is composed of 12 separately compiled modules. The order imposed by the Simula system is that any external parts used in a module have to be compiled prior to the compilation of the module which uses them. Therefore the modules are arranged in a linear hierarchy. For the same reason some parts, that contextually belong to certain modules, had to be moved to other modules.

Each module, except the highest one, is a class. Each class-module, except the lowest one, is a subclass of the next lower module. The highest module is a program whose main block is prefixed with the highest class-module name. In this way every module has access to all variables, classes and procedures defined in all the modules below it.

B.2 THE MODULES

The following sections list the modules and their functions in the system's hierarchical order.

B.2.1 TEXT HANDLING

This module includes basic text handling utilities which scan texts for certain characters or strings of characters, concatenate texts, turn characters and numbers into texts and modify texts by deleting or inserting subtexts.

B.2.2 INPUT / OUTPUT

Includes file handling routines which obtain file names from the user, create, open and close the various external files the system works with. Further it includes input and output routines which read from - or write to - the user's terminal or external files.

B.2.3 SEMANTICS

This module contains the class SEMANTIC which is the master class for all semantics. It contains all the procedures which are common to all semantic classes and the declaration of a virtual procedure SEM. Different semantic routines are obtained by declaring subclasses of SEMANTIC and writing the particular procedure SEM there. Objects of subclasses of SEMANTIC are used in the dictionary: Every defining dictionary element points (indirectly) to an object of the subclass which contains the appropriate SEM procedure. Objects of subclasses of SEMANTIC are also used as the nodes in the parsing tree.

Subclasses of SEMANTIC are declared in various modules of the system according to the location in the module hierarchy where they belong. Four of these subclasses are declared in the semantics module itself: SEMTERM, SEMSTR, SEMID and PRESEMSUBST. The first three are semantics of terminal parts of speech, strings and identifiers respectively. The fourth class contains the attributes of the semantics of text substitution (used in translation pass) which have to be known to modules located below the translator module. The class SEMSUBST is declared as a subclass of PRESEMSUBST in the translator module because its SEM procedure interacts recursively with the translation routine.

Other main parts in this module are the class of branches of the parsing tree and the class of number lists (which is used in PRESEMSUBST).

B.2.4 DICTIONARY

Includes the part of speech table, the dictionaries and related procedures.

The parts of speech are arranged in a hash table which consists of an array of 96 lists, one for each non-control character. The parts of speech are hashed according to the first character.

The dictionary is the data structure used by the parser. It is arranged in a binary tree. The attributes of each node are a part of speech, a definition and two links: NEXT and ALT. The definition and the links may be empty. The definition attribute points to a list of objects of the class DEFINITION (also declared in this module), which includes pointers to the left hand side part of speech of the syntax rule and to an object of a subclass of SEMANTIC. The NEXT link points to the node containing the next part of speech of the syntax rule. The ALT link points to a node whose part of speech may be used instead of this node's part of speech in order to obtain another rule.

Two dictionaries are used in the system: The definition dictionary, which is used in the syntax pass to parse the definitions entered by the user, and the user's dictionary, which is built in the syntax pass and used in the tranlsation pass.

In order to reduce parsing time in the syntax pass, the definition dictionary is divided into 4 sections. One section includes syntax rules that are used in all parts of a definition. The other three include rules that are used only for processing of a left-hand side text, declarations or right-hand side text of a definition. Whenever one of these parts of a definition is processed, the corresponding dictionary section is linked to the common section. In this way the dictionary size, and hence - the number of spurious arcs created by the parser, is kept to a minimum.

The MERGE procedure puts new rules into the dictionaries. It is used in the initialization step to build the definitions dictionary and in syntax pass to build the user's dictionary. It is this procedure that takes care of the order of priority discussed in chapter 6.

B.2.5 PARSING

The module includes the parser and all the related procedures.

The system permits the designer to use any language without syntax restrictions. In order to meet this requirement, a general parser has to be used. The parser used here is the bottom - up, right to left parser that has been successfully used in the REL system [35].

The parsing procedure is an attribute of the class of arcs: ARC. ARC objects have four data attributes: A pointer to a part of speech, a pointer to an object of a subclass of SEMANTIC, a NEXT link, which points to the next arc in the parsing graph (horizontal direction), and an ALT link, which points to an alternate arc (vertical direction).

Other parts that participate in the parsing process are the constituent stack (class CONSTITUENT) and the APPLY procedure. The parser pushes arcs onto the stack and, whenever a rule should be applied, the APPLY procedure is called in order to build a new arc around the arcs that are in the stack. Whenever a new arc is created, all its attributes are set up, in particular a new semantic object is created and linked to the semantics of the arc's constituents.

When parsing is completed, the ROOT procedure is called to find the root of the parsing tree. It is this routine that performs the algorithm described in chapter 7 in the translation pass.

Other important procedures in this module are: ARCSTRING which creates the initial string of arcs from the text to be parsed; prescanning procedures, which are attributes of ARC and are called prior to parsing in order to build arcs around strings (in syntax pass) and around numbers and identifiers (in both passes); PMRKR, OUTPOS and TAKEALOOK are used in debug mode to look at the parsing graph.

B.2.6 PARAMETER TABLE

When a definition is processd, its parameters are recorded in a table together with the parts of speech they represent and with information about the equality of different parameters. The parameter table (class PARTAB) and its related procedures and data structures are declared in this module. The module also includes the procedures GETRULES and BUILDRULES (both attributes of PARTAB) which use the information in the table in order to build the input to the MERGE procedure (in the dictionary module) and then call it to insert the rules into the user's dictionary.

B.2.7 SEMANTICS OF DEFINITIONS

The module includes subclasses of SEMANTIC containing the semantic routines which correspond to the syntax rules of definitions. Further it contains the procedure GETMACSYNTAX which builds the definitions dictionary. It uses the external file SYNTAX.MAC to read the syntax rules and the information about the dictionary section into which each rule belongs (see B.2.4) and the semantic routine to associate with each rule.

B.2.8 TRANSLATION

contains procedures and classes used in translation pass. The procedure TRANSLATE takes a text, passes it to the language processor, obtains its translation and repeats the process until the text remains unchanged. Class SEMSUBST is declared in this module as a subclass of PRESEMSUBST (see B.2.3). Its SEM procedure is the one that performs the translation. The class had to be declared in this module since, whenever a subtext that parsed to <STMT> is encountered, SEM calls TRANSLATE to translate the subtext and only afterwards resumes its own processing (see paragraph 9.3).

Other procedures and classes in this module read the input text and prescan it in order to parse its parameters according to the procedure's declarations.

B.2.9 DEFINITIONS PROCESSING

The simplest way to process the definitions is to pass a whole definition to the language processor and let it do the work. The drawback of this method is that in this way it is very difficult to put useful information about error locations into syntax-error messages. The messages have to be of a rather general nature, they can say that "there is something wrong with this input". The user himself has to do the job, which is quite laborious for long inputs, of pinpointing the error.

In order to overcome this problem, at least partially, the processing of definitions in this system is partially keyword and partially syntax driven. Keywords are used in order to divide macros and procedures into their main parts: Left-hand side, definition lines and right-hand side. Each part is then processed separately by the language processor. Syntax-error messages can restrict the error to the part being processed and make it easier for the user to locate it.

The module contains the procedures which read macros and procedures - interactively or from an external file - recognize their parts and pass them to the language processor.

The reason for putting this module above the translation module level is that it has to know about class SEMSUBST (semantics used in translation). Objects of this class have to be passed to the procedures which merge the defined syntax rules into the user's dictionary.

B.2.10 DICTIONARY DUMPING AND UPDATING

This module contains two procedures: DICDMP and UPDATE. DICDMP is called to display the dictionary contents on the terminal whenever one of the command-language commands DICTIONARY or PRIMITIVES is issued. UPDATE is called whenever the UPDATE command is issued from command-language level.

The two procedures had to be raised to this level because they use information from objects of class SEMSUBST (see B.2.8), which are attributes of every defining node in the user's dictionary.

B.2.11 SYSTEM DRIVERS

Includes two procedures: GETSYNTAX and GETPROCS. They are called whenever one of the commands SYNTAX or TRANSLATE, respectively, are issued from command-language level. GETSYNTAX drives the system in syntax pass. It reads the input - interactively or from an external file - and wnenever a macro or a procedure is encountered, the appropriate procedures are called to process it. GETPROCS performs a similar task in the translation pass.

B.2.12 COMMAND LANGUAGE

This module is a program which intializes the system by building the definitions dictionary, saves the core image in a file named SDS.EXE and then reads command language commands from the terminal and performs them by calling the appropriate procedures.

The file SDS.EXE can be used in order to run the system without having to wait for dictionary initialization and other initialization steps. If the command RUN SDS is issued from monitor level, the system resumes at the point where the file SDS.EXE was created and immediately is ready to accept command-language commands.

Appendix C

AN EXAMPLE

The following example consists of an SDS design of the problem of computing the 1000 first prime numbers. This problem appears as an example of stepwise program composition in Dijkstra's "Notes on Structured Programming" [10] and the SDS design presented here roughly follows Dijkstra's refinements.

The design (starting page 152) is almost self explanatory. Its bottom level consists of Simula constructs. Note that the refinement of INITIALIZE AUXILIARY VARIABLES which appears on the right-hand side of macro 8 was postponed almost to the end (macro 114) because only after completing all the refinements did I know what local variables were needed.

The design mixes Simula constructs with higher level constructs in some of the refinements. Successful parsing of these mixtures was achieved by using a dictionary with primitives of a subset of Simula as the starting point for the syntax pass. The file that was used to build this dictionary is on page 156.

The set of macros titled "High Level Boolean Expressions" (line 145) is not really necessary for the translation into Simula, but if a lower level target language is introduced, then these macros are needed in order to avoid confusing the high level constructs with Simula constructs due to their mixture in one statement in macro 70 (see chapter 8 for an explanation of this phenomenon).

After obtaining a Simula program which ran successfully (page 166), I ran the same design on a dictionary that translates a subset of Simula into Macro Assembly Language (instead of the dictionary with Simula primitives). In order to obtain a valid assembly language program I had to take care of the initialization and termination instructions that have to be in any such program, and therefore added initialization and termination statements to the procedure from line 1 which became:

```
PROC
MAIN PROGRAM
-->
INITIALIZE PRIME;
LIST 100 FIRST PRIMES;
TERMINATE PRIME
PEND
```

The file from which the Simula-Assembly translation dictionary was constructed starts on page 167. This file contains the refinements of the initialization and termination statements as well.

The resulting program, which ran successfully, starts on page 179.

Design of the Program: List 1000 First Primes

```
PROC
MAIN PROGRAM
-->
LIST 1000 FIRST PRIMES
PEND


MACRO
LIST 'N' FIRST PRIMES
WHERE
N:<NU>
-->
INITIALIZE TABLE OF SIZE 'N';
INITIALIZE AUXILIARY VARIABLES;
FILL TABLE WITH 'N' FIRST PRIMES;
PRINT 'N' SIZED TABLE AND TERMINATE
MEND


MACRO
<BEGINSTMT>: INITIALIZE 'ARRAY' OF SIZE 'N'
WHERE
N:<NU>
-->
BEGIN
INTEGER ARRAY 'ARRAY'[1:'N']
MEND


MACRO
FILL TABLE WITH 'N' FIRST PRIMES
WHERE
N:<NU>
-->
TABLE[1]:=2;  TABLE[2]:=3;
FOR III:=3 STEP 1 UNTIL 'N' DO
BEGIN
    MAKE TABLE[III] THE III"TH PRIME
END
MEND


```

```
MACRO
MAKE 'ENTRY'['I'] THE 'I'"TH PRIME
WHERE
I:<ID>
-->
JJJ:='ENTRY'['I'-1];
JPRIME:=FALSE;
WHILE NOT JPRIME DO
BEGIN
    JJJ:=JJJ+2;
    JPRIME:=JJJ IS A PRIME;
END;
    'ENTRY'['I']:=JJJ
MEND



MACRO
'B':='N' IS A PRIME
WHERE
N:<ID>,<NU>
-->
COMPUTE UPPER BOUND ORD FOR THIS 'N';
'B':=TRUE;
KKK:=0;
FOR KKK:=KKK+1 WHILE 'B' AND KKK<ORD+1 DO
BEGIN
    'B':=TABLE[KKK] IS NOT A FACTOR OF 'N'
END
MEND


MACRO
COMPUTE UPPER BOUND ORD FOR THIS 'X'
WHERE
X: <ID>,<NU>
-->
WHILE SQUARE<'X'+1 DO
BEGIN
    ORD:=ORD+1;
    SQUARE:=TABLE[ORD]*TABLE[ORD];
END
MEND


MACRO
<HBOOEX>: 'A' IS NOT A FACTOR OF 'B'
WHERE
A,B:<HID>,<HNU>
-->
NOT ('B'-('B'//'A')*'A'=0)
MEND
```

```


MACRO
<ENDSTMT>: PRINT 'N' SIZED 'TABLE' AND TERMINATE
WHERE
N:<NU>
-->
KKK:='N'-10;
FOR III:=0 STEP 10 UNTIL KKK DO
BEGIN
    FOR JJJ:=1 STEP 1 UNTIL 10 DO
       OUTINT('TABLE'[III+JJJ],8);
    OUTIMAGE;
END;
END
MEND


MACRO
INITIALIZE AUXILIARY VARIABLES
-->
DECLARE AUXILIARY VARIABLES;
SET THEIR INITIAL VALUES
MEND


MACRO
DECLARE AUXILIARY VARIABLES
-->
INTEGER III,JJJ,KKK,ORD,SQUARE;
BOOLEAN JPRIME
MEND


MACRO
SET THEIR INITIAL VALUES
-->
ORD:=1;  SQUARE:=4
MEND


!   High Level Boolean Expressions

MACRO
<HBOOEX>:NOT 'HB'
WHERE
HB:<HBOOEX>
-->
PRIMITIVE
MEND


```

```
MACRO
<HBOOEX>: ('HB')
WHERE
HB:<HBOOEX>
-->
PRIMITIVE
MEND


MACRO
'Y':='HB'
WHERE
HB:<HBOOEX>
-->
PRIMITIVE
MEND


MACRO
<HID>:'ID'['I']
WHERE
I:<ID>,<NU>
-->
PRIMITIVE
MEND


MACRO
<HNU>: 'N'
WHERE
N: <NU>
-->
PRIMITIVE
MEND


MACRO
<HID>:'ID'
-->
PRIMITIVE
MEND
```

File of Simula Primitives

```
!    Block Structure

MACRO
BEGIN 'SL' END
WHERE
SL:<STMTL>
-->
PRIMITIVE
MEND


MACRO
BEGIN 'SL'; END
WHERE
SL:<STMTL>
-->
PRIMITIVE
MEND


MACRO
BEGIN 'DL'; 'SL' END
WHERE
DL:<DECL>  SL:<STMTL>
-->
PRIMITIVE
MEND


MACRO
BEGIN 'DL'; 'SL'; END
WHERE
DL:<DECL>  SL:<STMTL>
-->
PRIMITIVE
MEND


!    Program Structure

MACRO
<BEGINSTMT>: BEGIN 'DL'
WHERE
DL: <DECL>
-->
PRIMITIVE
MEND


```

```
MACRO
<ENDSTMT>: 'SL' END
WHERE
SL: <STMTL>
-->
PRIMITIVE
MEND


MACRO
<ENDSTMT>: 'SL'; END
WHERE
SL: <STMTL>
-->
PRIMITIVE
MEND


MACRO
'A'; 'SL'; 'Z'
WHERE
A:<BEGINSTMT>  SL:<STMTL>  Z:<ENDSTMT>
-->
PRIMITIVE
MEND


!   Statement List

MACRO
<STMTL>: 'S'
WHERE
S: <STMT>
-->
PRIMITIVE
MEND


MACRO
<STMTL>: 'SL'; 'S'
WHERE
SL:<STMTL>  S:<STMT>
-->
PRIMITIVE
MEND
```

```
!   Declarations

MACRO
<DEC>: 'T' 'IDL'
WHERE
T:<TYPE>  IDL:<IDLIST>
-->
PRIMITIVE
MEND


MACRO
<DEC>: 'T' ARRAY 'ID'['N1':'N2']
WHERE
T:<TYPE>   ID:<ID>   N1,N2:<NU>
-->
PRIMITIVE
MEND


MACRO
<IDLIST>: 'ID'
WHERE
ID:<ID>
-->
PRIMITIVE
MEND


MACRO
<IDLIST>: 'IDL','ID'
WHERE
IDL:<IDLIST>  ID:<ID>
-->
PRIMITIVE
MEND


MACRO
<DECL>: 'D'
WHERE
D:<DEC>
-->
PRIMITIVE
MEND


```

```
MACRO
<DECL>: ’DL’; ’D’
WHERE
DL:<DECL>  D:<DEC>
-->
PRIMITIVE
MEND


MACRO
<TYPE>: INTEGER
-->
PRIMITIVE
MEND


MACRO
<TYPE>: BOOLEAN
-->
PRIMITIVE
MEND


MACRO
<ID>: ’A’[’E’]
WHERE
E:<AEX>
-->
PRIMITIVE
MEND


!   Loops

MACRO
FOR ’I’:=’INIT’ STEP ’INC’ UNTIL ’FINAL’ DO ’S’
WHERE
INIT,INC,FINAL:<AEX>   S:<STMT>
-->
PRIMITIVE
MEND


MACRO
WHILE ’COND’ DO ’S’
WHERE
COND:<BOOEX> S:<STMT>
-->
PRIMITIVE
MEND


```

```
MACRO
FOR ’I’:=’EX’ WHILE ’BEX’ DO ’S’
WHERE
EX:<AEX>  BEX:<BOOEX>  S:<STMT>
-->
PRIMITIVE
MEND


!   Assignment Statements

MACRO
’Y’:=’EX’
WHERE
EX:<AEX>
-->
PRIMITIVE
MEND


MACRO
’Y’:=’B’
WHERE
B:<BOOEX>
-->
PRIMITIVE
MEND


!   Boolean Expressions

MACRO
<BOOEX>: ’A’<’B’
WHERE
A,B:<AEX>
-->
PRIMITIVE
MEND


MACRO
<BOOEX>: ’A’>’B’
WHERE
A,B:<AEX>
-->
PRIMITIVE
MEND
```

```
MACRO
<BOOEX>:’A’=’B’
WHERE
A,B:<AEX>
-->
PRIMITIVE
MEND


MACRO
<BOOEX>: NOT ’B’
WHERE
B: <BOOEX>
-->
PRIMITIVE
MEND


MACRO
<BOOEX>: TRUE
-->
PRIMITIVE
MEND


MACRO
<BOOEX>: FALSE
-->
PRIMITIVE
MEND


MACRO
<BOOEX>: ’B’
-->
PRIMITIVE
MEND


MACRO
<BOOEX>: (’B’)
WHERE
B:<BOOEX>
-->
PRIMITIVE
MEND
```

```
MACRO
<BOOEX>: 'B1' AND 'B2'
WHERE
B1,B2:<BOOEX>
-->
PRIMITIVE
MEND


MACRO
<BOOEX>: 'B1' OR 'B2'
WHERE
B1,B2:<BOOEX>
-->
PRIMITIVE
MEND


!   Arithmetic Expressions

MACRO
<AEX>: ABS('X')
WHERE
X:<AEX>
-->
PRIMITIVE
MEND


MACRO
<AEX>:'S'
WHERE
S:<SUM>
-->
PRIMITIVE
MEND


MACRO
<SUM>:'T'
WHERE
T:<TERM>
-->
PRIMITIVE
MEND


```

```
MACRO
<SUM>:'S''A''T'
WHERE
S:<SUM>  A:<AOP>  T:<TERM>
-->
PRIMITIVE
MEND


MACRO
<TERM>:'S'
WHERE
S:<SECONDARY>
-->
PRIMITIVE
MEND


MACRO
<TERM>:'T''M''S'
WHERE
T:<TERM>  M:<MOP>   S:<SECONDARY>
-->
PRIMITIVE
MEND


MACRO
<SECONDARY>:+'S'
WHERE
S:<SECONDARY>
-->
PRIMITIVE
MEND


MACRO
<SECONDARY>:-'S'
WHERE
S:<SECONDARY>
-->
PRIMITIVE
MEND


MACRO
<SECONDARY>:'P'
WHERE
P:<PRIMARY>
-->
PRIMITIVE
MEND
```

```


MACRO
<PRIMARY>:'N'
WHERE
N:<NU>
-->
PRIMITIVE
MEND


MACRO
<PRIMARY>:'ID'
WHERE
ID:<ID>
-->
PRIMITIVE
MEND


MACRO
<PRIMARY>:('S')
WHERE
S:<SUM>
-->
PRIMITIVE
MEND


!   Binary Arithmetic Operators

MACRO
<AOP>:+
-->
PRIMITIVE
MEND


MACRO
<AOP>:-
-->
PRIMITIVE
MEND


MACRO
<MOP>:*
-->
PRIMITIVE
MEND


```

```
MACRO
<MOP>:/
-->
PRIMITIVE
MEND


MACRO
<MOP>://
-->
PRIMITIVE
MEND


!   Output Statements

MACRO
OUTIMAGE
-->
PRIMITIVE
MEND


MACRO
OUTINT('A','N')
WHERE
A:<AEX>    N:<NU>
-->
PRIMITIVE
MEND
```

The Simula Program that SDS Turned out

```
BEGIN
INTEGER ARRAY TABLE[1:1000];
INTEGER III,JJJ,KKK,ORD,SQUARE;
BOOLEAN JPRIME;
ORD:=1;  SQUARE:=4;
TABLE[1]:=2;  TABLE[2]:=3;
FOR III:=3 STEP 1 UNTIL 1000 DO
BEGIN
    JJJ:=TABLE[III-1];
JPRIME:=FALSE;
WHILE NOT JPRIME DO
BEGIN
    JJJ:=JJJ+2;
    WHILE SQUARE<JJJ+1 DO
BEGIN
    ORD:=ORD+1;
    SQUARE:=TABLE[ORD]*TABLE[ORD];
END;
JPRIME:=TRUE;
KKK:=0;
FOR KKK:=KKK+1 WHILE JPRIME AND KKK<ORD+1 DO
BEGIN
    JPRIME:=NOT (JJJ-(JJJ//TABLE[KKK])*TABLE[KKK]=0)
END;
END;
    TABLE[III]:=JJJ
END;
KKK:=1000-10;
FOR III:=0 STEP 10 UNTIL KKK DO
BEGIN
    FOR JJJ:=1 STEP 1 UNTIL 10 DO
       OUTINT(TABLE[III+JJJ],8);
    OUTIMAGE;
END;
END
```

File of Simula-to-Assembly Translation Rules

```
!   Block Structure

MACRO
BEGIN 'SL' END
WHER   6   SL:<STMTL>
-->
'SL'
MEND


MACRO
BEGIN 'SL'; END
WHERE
SL:<STMTL>
-->
'SL'
MEND


MACRO
BEGIN 'DL'; 'SL' END
WHERE
DL:<DECL>  SL:<STMTL>
-->
'DL'
'SL'
MEND


MACRO
BEGIN 'DL'; 'SL'; END
WHERE
DL:<DECL>  SL:<STMTL>
-->
'DL'
'SL'
MEND


!   Program Structure

MACRO
<BEGINSTMT>: BEGIN 'DL'
WHERE
DL: <DECL>
-->
'DL'
MEND
```

```


MACRO
<ENDSTMT>: 'SL' END
WHERE
SL: <STMTL>
-->
'SL'
MEND


MACRO
<ENDSTMT>: 'SL'; END
WHERE
SL: <STMTL>
-->
'SL'
MEND


MACRO
'A'; 'SL'; 'Z'
WHERE
A:<BEGINSTMT>  SL:<STMTL>   Z:<ENDSTMT>
-->
PRIMITIVE
MEND


!   Statement List

MACRO
<STMTL>: 'S'
WHERE
S: <STMT>
-->
PRIMITIVE
MEND


MACRO
<STMTL>: 'SL'; 'S'
WHERE
SL:<STMTL>  S:<STMT>
-->
'SL'
'S'
MEND


```

```
!   Declarations

MACRO
<DEC>: 'T' 'ID'
WHERE
T:<TYPE>
-->
          INTEGER 'ID'
          MOVEM   6,'ID'
MEND


MACRO
<DEC>: 'D','ID'
WHERE
D:<DEC>
-->
'D'
          INTEGER 'ID'
          MOVEM   6,'ID'
MEND


MACRO
<DEC>:'ADEC'
WHERE
ADEC:<ADEC>
-->
PRIMITIVE
MEND


MACRO
<ADEC>: 'T' ARRAY 'ID'['N1':'N2']
WHERE
T:<TYPE>   ID:<ID>   N1,N2:<NU>
-->
          ARRAY   'ID'[^D'N2'-^D'N1'+2]
          MOVE    17,[^D'N1'-1]
          MOVEM   17,'ID'
MEND


MACRO
<DECL>: 'D'
WHERE
D:<DEC>
-->
PRIMITIVE
MEND


```

```
MACRO
<DECL>: 'DL'; 'D'
WHERE
DL:<DECL>  D:<DEC>
-->
'DL'
'D'
MEND


MACRO
<TYPE>: INTEGER
-->
PRIMITIVE
MEND

MACRO
<TYPE>:BOOLEAN
-->
PRIMITIVE
MEND


!   Loops

MACRO
FOR 'I':='INIT' STEP 'INC' UNTIL 'FINAL' DO 'S'
WHERE
I:<AID>   INIT,INC,FINAL:<AEX>   S:<STMT>
-->
'INIT'
'I'
        POP     A,@5
$FORS$:'FINAL'
        POP     A,13
'I'
        SUB     13,@5
        JUMPL   13,$EFORS$
'S'
'INC'
        POP     A,13
'I'
        ADDM    13,@5
        JRST    $FORS$
$EFORS$:
MEND


```

```
MACRO
WHILE 'COND' DO 'S'
WHERE
COND:<BOOEX> S:<STMT>
-->
$WH$:  'COND'
        JUMPE    7,$EWH$
'S'
        JRST    $WH$
$EWH$:
MEND


MACRO
FOR 'I':='EX' WHILE 'BEX' DO 'S'
WHERE
I:<AID>  EX:<AEX>  BEX:<BOOEX>  S:<STMT>
-->
$FORW$:'EX'
'I'
        POP     A,@5
'BEX'
        JUMPE    7,$EFORW$
'S'
        JRST    $FORW$
$EFORW$:
MEND


!   Array Identifier

MACRO
<AID>: 'A'['E']
WHERE
E:<AEX>
-->
'E'
        POP     A,5
        SUB     5,'A'
        ADDI    5,'A'
MEND


MACRO
<AID>:'X'
-->
        MOVEI   5,'X'
MEND


```

```
!   Assignment Statements

MACRO
'Y':='EX'
WHERE
Y:<AID>   EX:<AEX>
-->
'EX'
'Y'
        POP     A,@5
MEND


MACRO
'Y':='B'
WHERE
Y:<AID>   B:<BOOEX>
-->
'B'
'Y'
        MOVEM    7,@5
MEND


!   Boolean Expressions

MACRO
<BOOEX>: ('B')
WHERE
B:<BOOEX>
-->
'B'
MEND


MACRO
<BOOEX>: 'A'<'B'
WHERE
A,B:<AEX>
-->
'A'
'B'
        POP     A,14
        POP     A,12
        SUB     14,12
        SETO     7,
        SKIPG   14
        SETZ     7,
MEND

```

```
MACRO
<BOOEX>: 'A'='B'
WHERE
A,B: <AEX>
-->
'A'
'B'
        POP     A,14
        POP     A,12
        SUB     14,12
        SETO     7,
        SKIPE   14
        SETZ     7,
MEND


MACRO
<BOOEX>: 'B1' AND 'B2'
WHERE
B1,B2:<BOOEX>
-->
'B1'
        MOVE    10,7
'B2'
        AND      7,10
MEND


MACRO
<BOOEX>: NOT 'B'
WHERE
B: <BOOEX>
-->
'B'
        SETCA    7,7
MEND


MACRO
<BOOEX>: 'B'
-->
        MOVE     7,'B'
MEND


MACRO
<BOOEX>: TRUE
-->
        SETO     7,
MEND


```

```
MACRO
<BOOEX>: FALSE
-->
          SETZ      7,
MEND



!   Arithmetic Expressions

MACRO
<AEX>:'S'
WHERE
S:<SUM>
-->
PRIMITIVE
MEND


MACRO
<SUM>:'T'
WHERE
T:<TERM>
-->
PRIMITIVE
MEND


MACRO
<SUM>:'S''A''T'
WHERE
S:<SUM>  A:<AOP>  T:<TERM>
-->
'T'
'S'
          POP       A,12
          'A'       12,(A)
MEND


MACRO
<TERM>:'S'
WHERE
S:<SECONDARY>
-->
PRIMITIVE
MEND


```

```
MACRO
<TERM>:’T’’M’’S’
WHERE
T:<TERM>  M:<MOP>    S:<SECONDARY>
-->
’S’
’T’
        POP     A,12
        ’M’   12,(A)
MEND


MACRO
<SECONDARY>:+’S’
WHERE
S:<SECONDARY>
-->
’S’
MEND


MACRO
<SECONDARY>:’P’
WHERE
P:<PRIMARY>
-->
PRIMITIVE
MEND


MACRO
<PRIMARY>:’N’
WHERE
N:<NU>
-->
        PUSH    A,[^D’N’]
MEND


MACRO
<PRIMARY>:’ID’
-->
        PUSH    A,’ID’
MEND


```

```
MACRO
<PRIMARY>:'AID'
WHERE
AID:<AID>
-->
'AID'
        PUSH    A,@5
MEND


MACRO
<PRIMARY>:('S')
WHERE
S:<SUM>
-->
'S'
MEND


MACRO
<AEX>: ININT
-->
        MOVEI   1,.PRIIN
        MOVEI   3,12
        NIN
        ERJMP   NINER
        PUSH    A,2
MEND


!   Binary Arithmetic Operators

MACRO
<AOP>:+
-->
ADDM
MEND


MACRO
<AOP>:-
-->
SUBM
MEND


MACRO
<MOP>:*
-->
IMULM
MEND

```

```
MACRO
<MOP>:/
-->
IDIVM
MEND


MACRO
<MOP>://
-->
IDIVM
MEND


!   Output Statements

MACRO
OUTIMAGE
-->
        HRROI   1,[ASCIZ/
/]
        PSOUT
MEND


MACRO
OUTINT('A','N')
WHERE
A:<AEX>   N:<NU>
-->
'A'
        MOVEI   1,.PRIIN
        POP     A,2
        MOVEI   3,12
        HRLI    3,100K+^D'N'
        NOUT
        ERJMP   NOUTER
MEND


!   Program Initialization

MACRO
INITIALIZE 'NAME'
-->
        SEARCH MONSYM
        A=4
'NAME':   MOVE A,[IOWD 1000,STACK]
        SETZ    6,
MEND
```

```

!   Program Termination

MACRO
TERMINATE 'NAME'
-->
        HALTF
NINER:  HRROI   1,[ASCIZ/
ININT ERROR
/]
        PSOUT
        HALTF
NOUTER: HRROI   1,[ASCIZ/
OUTINT ERROR
/]
        PSOUT
        HALTF
STACK:  BLOCK   1000
        END 'NAME'
MEND
```

The Assembly Language Program that SDS Turned out

```
SEARCH MONSYM
        A=4
PRIME:   MOVE A,[IOWD 1000,STACK]
        SETZ    6,
ARR             MOVE    17,[^D1-1]
        MOVEM   17,TABLE
INTEGER III
        MOVEM   6,III
        INTEGER JJJ
        MOVEM   6,JJJ
        INTEGER KKK
        MOVEM   6,KKK
        INTEGER ORD
        MOVEM   6,ORD
        INTEGER SQUARE
        MOVEM   6,SQUARE
INTEGER JPRIME
        MOVEM   6,JPRIME
PUSH    A,[^D1]
MOVEI   5,ORD
        POP     A,@5
PUSH    A,[^D4]
MOVEI   5,SQUARE
        POP     A,@5
PUSH    A,[^D2]
PUSH    A,[^D1]
        POP     A,5
        SUB     5,TABLE
        ADDI    5,TABLE
        POP     A,@5
PUSH    A,[^D3]
PUSH    A,[^D2]
        POP     A,5
        SUB     5,TABLE
        ADDI    5,TABLE
        POP     A,@5
PUSH    A,[^D3]
MOVEI   5,III
        POP     A,@5
FORS2:PUSH    A,[^D1000]
        POP     A,13
MOVEI   5,III
        SUB     13,@5
        JUMPL   13,EFORS2
PUSH    A,[^D1]
PUSH    A,III
        POP     A,12
        SUBM    12,(A)
```

```
        POP     A,5
        SUB     5,TABLE
        ADDI    5,TABLE
        PUSH    A,@5
MOVEI   5,JJJ
        POP     A,@5
SETZ     7,
MOVEI   5,JPRIME
        MOVEM    7,@5
WH3:  MOVE     7,JPRIME
        SETCA    7,7
        JUMPE    7,EWH3
PUSH    A,[^D2]
PUSH    A,JJJ
        POP     A,12
        ADDM    12,(A)
MOVEI   5,JJJ
        POP     A,@5
WH4:  PUSH    A,SQUARE
PUSH    A,[^D1]
PUSH    A,JJJ
        POP     A,12
        ADDM    12,(A)
        POP     A,14
        POP     A,12
        SUB     14,12
        SETO     7,
        SKIPG   14
        SETZ     7,
        JUMPE    7,EWH4
PUSH    A,[^D1]
PUSH    A,ORD
        POP     A,12
        ADDM    12,(A)
MOVEI   5,ORD
        POP     A,@5
PUSH    A,ORD
        POP     A,5
        SUB     5,TABLE
        ADDI    5,TABLE
        PUSH    A,@5
PUSH    A,ORD
        POP     A,5
        SUB     5,TABLE
        ADDI    5,TABLE
        PUSH    A,@5
        POP     A,12
        IMULM   12,(A)
MOVEI   5,SQUARE
        POP     A,@5
        JRST    WH4
EWH4:
```

```
SETO     7,
MOVEI   5,JPRIME
        MOVEM    7,@5
PUSH    A,[^D0]
MOVEI   5,KKK
        POP     A,@5
FORW5:PUSH    A,[^D1]
PUSH    A,KKK
        POP     A,12
        ADDM    12,(A)
MOVEI   5,KKK
        POP     A,@5
MOVE     7,JPRIME
        MOVE    10,7
PUSH    A,KKK
PUSH    A,[^D1]
PUSH    A,ORD
        POP     A,12
        ADDM    12,(A)
        POP     A,14
        POP     A,12
        SUB     14,12
        SETO     7,
        SKIPG   14
        SETZ     7,
        AND      7,10
        JUMPE    7,EFORW5
PUSH    A,KKK
        POP     A,5
        SUB     5,TABLE
        ADDI    5,TABLE
        PUSH    A,@5
PUSH    A,KKK
        POP     A,5
        SUB     5,TABLE
        ADDI    5,TABLE
        PUSH    A,@5
PUSH    A,JJJ
        POP     A,12
        IDIVM   12,(A)
        POP     A,12
        IMULM   12,(A)
PUSH    A,JJJ
        POP     A,12
        SUBM    12,(A)
PUSH    A,[^D0]
        POP     A,14
        POP     A,12
        SUB     14,12
        SETO     7,
        SKIPE   14
        SETZ     7,
```

```
          SETCA     7,7
MOVEI     5,JPRIME
          MOVEM      7,@5
          JRST      FORW5
EFORW5:
          JRST      WH3
EWH3:
MOVE       7,JJJ
PUSH      A,III
          POP       A,5
          SUB       5,TABLE
          ADDI      5,TABLE
          MOVEM      7,@5
PUSH      A,[^D1]
          POP       A,13
MOVEI     5,III
          ADDM      13,@5
          JRST      FORS2
EFORS2:
PUSH      A,[^D10]
PUSH      A,[^D1000]
          POP       A,12
          SUBM      12,(A)
MOVEI     5,KKK
          POP       A,@5
PUSH      A,[^D0]
MOVEI     5,III
          POP       A,@5
FORS0:PUSH     A,KKK
          POP       A,13
MOVEI     5,III
          SUB       13,@5
          JUMPL     13,EFORS0
PUSH      A,[^D1]
MOVEI     5,JJJ
          POP       A,@5
FORS1:PUSH     A,[^D10]
          POP       A,13
MOVEI     5,JJJ
          SUB       13,@5
          JUMPL     13,EFORS1
PUSH      A,JJJ
PUSH      A,III
          POP       A,12
          ADDM      12,(A)
          POP       A,5
          SUB       5,TABLE
          ADDI      5,TABLE
          PUSH      A,@5
          MOVEI     1,.PRIIN
          POP       A,2
          MOVEI     3,12
```

```
        HRLI    3,100K+^D8
        NOUT
        ERJMP   NOUTER
PUSH    A,[^D1]
        POP     A,13
MOVEI   5,JJJ
        ADDM    13,@5
        JRST    FORS1
EFORS1:
HRROI   1,[ASCIZ/
/]
        PSOUT
PUSH    A,[^D10]
        POP     A,13
MOVEI   5,III
        ADDM    13,@5
        JRST    FORS0
EFORS0:
HALTF
NINER:  HRROI   1,[ASCIZ/
ININT ERROR
/]
        PSOUT
        HALTF
NOUTER: HRROI   1,[ASCIZ/
OUTINT ERROR
/]
        PSOUT
        HALTF
STACK:  BLOCK   1000
        END PRIME
```

Bibliography

[1] Balzer B., Transformational Implementation: An Example, USC Information Sciences Institute, August 1979

[2] Basili V. R. and Turner A. J., "Iterative Enhancement A Practical Technique for Software Development", in IEEE Transactions on Software Engineering, Volume 1, Part 4, December 1975

[3] Bauer F. L., "Software Engineering", in Advanced Course on Software Engineering, Springer Verlag, 1973

[4] Bell T. E. and Bixler D. C. and Dyer M. E., "An Extendable Approach to Computer Aided Software Requirements Engineering", in IEEE Transactions on Software Engineering, Volume 3, Part 1, January 1977

[5] Brooks F. P., The Mythical Man Month, Addison-Wesley, 1975

[6] Caine S. H. and Gordon E. K., "PDL - A Tool for Software Design", in AFIPS, Volume 44, 1975

[7] Cheatham T. E. and Fisher A. and Jorrand P., "On the Basis of ELF - An Extensible Language Facility", in AFIPS, Volume 33, 1968

[8] Cheatham T. E., "The TGS-II Translator Generator System", in IFIP, Volume 2, 1965

[9] Dijkstra E. W., A Discipline of Programming, Prentice-Hall, 1976

[10] Dijkstra E. W., "Notes on Structured Programming", in Structured Programming, Academic Press, 1975

[11] Galler B. A. and Graham R. M., "The MAD Definition Facility", in Communications of the ACM, Volume 12, Part 8, August 1969

[12] Goos G., "Hierarchies", in Advanced Course in Software Engineering, Springer Verlag, 1973

[13] Harrison M. C., "BALM - An Extendable List-Processing Language", in AFIPS, Volume 36, 1970

[14] Hobbs J. R., From Well-Written Algorithm Descriptions into Code, Research Report, Department of Computer Science, City University of New York, July 1977

[15] Howden W. E., "DISSECT - A symbolic Evaluation and Program Testing System", in IEEE Transactions on Software Engineering, Volume 4, Part 1, January 1978

[16] Irons E. T., "Experience With an Extensible Language", in Communications of the ACM, Volume 13, Part 1, January 1970

[17] Jensen K. and Wirth N., Pascal, User Manual and Report, Springer Verlag, 1978

[18] King J. C., "Symbolic Execution and Program Testing", in Communications of the ACM, Volume 19, Part 7, July 1976

[19] Kleine H., Software Design and Documentation Language, JPL Publication 77-24, 1977

[20] Leavenworth B. M., "Syntax Macros and Extended Translation", in Communications of the ACM, Volume 9, Part 11, November 1966

[21] Leroy H., "A Macro-Generator for Algol", in AFIPS, Volume 30, 1967

[22] MacCallum I. R. and Morris D. and Rohl J. S. and Brooker R. A., "The Compiler Compiler" in Annual Review in Automatic Programming, 1963

[23] McIlroy M.D., "Macro Implementation Extensions of Compiler Languages", in Communications of the ACM, Volume 3, Part 4, April 1960

[24] Poole P. C. and Waite W. M., "Machine Independent Software", in Second Symposium on Operating Systems Principles, 1969

[25] Randall D. L., Formal Methods in the Foundations of Science, PhD Thesis, California Institute of Technology, 1970

[26] Row J. R., "Automatic Data Structure Selection: An Example and Overview", in Communications of the ACM Volume 21, Part 5, May 1978

[27] Rowe L. A. and Tonge F. M., "Automating the Selection of Implementation Structures", in <u>IEEE Transactions on Software Engineering</u>, Volume 4, Part 6, November 1978

[28] Schuman S. A. and Jorrand P., "Definition Mechanisms in Extensible Programming Languages", in <u>AFIPS</u>, Volume 37, 1970

[29] Solentseff N., "A Classification of Extensible Programming Languages", in <u>Information Processing Letters</u>, Volume 1, Part 3, February 1972

[30] Solentseff N. and Yezerski A., "A Survey of Extensible Programming Languages", in <u>Annual Review in Automatic Programming</u>, Volume 7, Part 5, 1974

[31] Solentseff N. and Yezerski A., "ECT - An Extensible Contractable Translator System", in <u>Information Processing Letters</u>, Volume 1, Part 3, February 1972

[32] Standish T. A., "Extensibility in Programming Language Design", in <u>AFIPS</u>, Volume 44, 1975

[33] Strong J. and Wegstein J., "The Problem of Program Communication with Changing Machines", in <u>CACM</u>, Volume 1, Parts 8 & 9, 1958

[34] Teichroew D. and Hershey E. A. III, "PSL/PSA: A Computer-Aided Technique for Structured Documentaion and Analysis of Information Processing Systems", in <u>IEEE Transactions on Software engineering</u>, Volume 3, Part 1, January 1977

[35] Thompson B. H. and Thompson F. B., "The REL System as Prototype", in <u>Advances in Computers</u>, Volume 13 Academic Press, 1975

[36] Wegbreit B., "The ECL Programming System", in <u>AFIPS</u>, Volume 39, 1971

[37] Wilkes M. V., "Software engineering and Structured Programming", in <u>IEEE Transactions on Software Engineering</u>, Volume 2, Part 4, December 1976

[38] Wirth N., "Program Development by Stepwise Refinement", in <u>Communications of the ACM</u>, Volume 14, Part 4, April 1971

[39] Yourdan E. and Constantine L. L., Structured Design, Yourdan Inc., February 1976

[40] "YACC - Yet Another Compiler Compiler", in Documents For Use With the UNIX Time-Sharing System, Sixth Edition, The Western Electric Company

[41] The WELLMADE System Design Methodology, Honeywell Publication, October 1979