

912

TRANSLATING 'C' PROGRAMS FOR EXECUTION ON PROLOG BASED SYSTEMS

Dissertation submitted to
Jawaharlal Nehru University
in partial fulfilment of the requirements
for the award of the Degree of

MASTER OF TECHNOLOGY
in
COMPUTER SCIENCE AND TECHNOLOGY

by
NEERAJ AGGARWAL

**SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110 067,
JANUARY 1992**

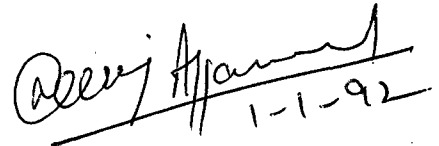
CERTIFICATE

This dissertation titled,

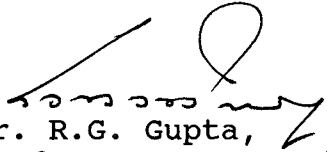
**TRANSLATING 'C' PROGRAMS FOR EXECUTION ON PROLOG
BASED MACHINES.**

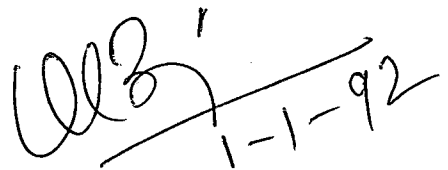
has been done by Mr. Neeraj Aggarwal, a bonafide student of School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi.

This work is original and has not been submitted for any degree or diploma in any other university or institute.


1-1-92

Neeraj Aggarwal


Dr. R.G. Gupta,
Professor & Dean, 21/1/92
SC&SS, J.N.U.,
New Delhi.


1-1-92

Dr. K.K. Bharadwaj
Professor,
SC&SS, J.N.U.,
New Delhi.

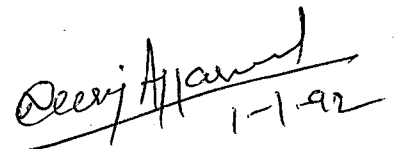
ACKNOWLEDGEMENT

I am grateful to my supervisor Prof. K. K. Bharadwaj who provided me the necessary insight into the subject and was always there with his helping hand in case of difficulties. But for his able guidance, it wouldn't have been possible for me to complete the project. He had the patience of listening to my problems and providing constructive suggestions.

I am thankful to Prof. R. G. Gupta for his constant inspiration and help. I also thank all my teachers who shared their valuable skills, expertise and time. They helped me realize the difficulties involved in making a transgression from the theoretical aspect to implementation aspect.

Beside this, I cannot forget to thank my colleagues and friends for their encouragement, suggestions and worthy discussions. They help in creating an atmosphere in which one is able to keep up his spirits.

Thanks are also due to all the people who directly or indirectly helped me in the due course of my work.


1-1-92

Neeraj Aggarwal

ABSTRACT

Developing a program that understands a input source program in 'C' language and then translate it into PROLOG rules by source-to-source translation is described here in this dissertation. We say a program understands a programming language if it behaves by taking a correct or acceptable action in response to the input. The behavior, or more explicitly the action taken in this program is only an internal response. Based on the syntax of the input program the action may simply be the creation of some internal data structures, or adding a missing token by simply comparing it with the grammar rule. The strategy followed is basically syntax directed translation scheme in top-down recursive parser. This translator is implemented using languages : PROLOG & 'C'.

Also, we explore many of the important issues related to radically different control and data structures in these two languages and the problems associated with translation process are also discussed here.

CONTENTS

ABSTRACT

ch 1	INTRODUCTION	1
1.1	Language Understanding	1
1.2	Objective and Scope of the Project	3
1.3	Motivation for Project Selection	4
ch 2	THEORETIC FOUNDATION OF TRANSLATOR	6
2.1	Programming Languages	6
2.2	Notations for Grammers	7
2.2.1	Classification of Grammers	8
2.3	Parsers	9
2.3.1	Top - Down Parser	10
2.3.2	Recusive - Decent Parsing	13
2.3.3	Transition Diagrams	14
2.4	Error Recovery	15
2.5	Syntax Directed Translation Scheme	16
ch 3	IMPERATIVE AND FUNCTIONAL LANGUAGES	18
3.1	Imperative Languages	18
3.2	Functional Languages	19
3.3	Prolog & 'C'	21
3.3.1	The 'C' Language	22
3.3.2	The language Prolog	23
3.3.3	Difference Between Prolog & 'C'	26

ch 4	SYSTEM ANALYSIS AND DESIGN	28
4.1	Translator	29
4.2	The Source Language	29
4.3	The Target Language	30
4.4	The Translation Scheme	31
4.4.1	Analysis of Source Program	32
4.4.2	Error Handling Technique	36
4.5	Translation Actions	40
4.5.1	Declarations	41
4.5.2	Statements	41
4.5.3	Common Data Structures	45
ch 5	CONCLUSION AND SUGGESTED ENHANCEMENTS	47
5.1	Conclusion	47
5.2	Suggested Enhancements	48
	BIBLIOGRAPHY	50
	APPENDICIES	
A	Grammer For SubC	
B	Transition Diagrams For SubC	
C	Test Results	

CHAPTER 1

INTRODUCTION

This chapter will provide a brief overview on the language understanding (programming & natural) and translation schemes. In addition, scope of the project and motivation for selecting this project is also discussed.

1.1 LANGUAGE UNDERSTANDING

When in 1950, Allan Turing wrote a controversial article entitled "Computing Machinery and Intelligence", he started with a question : "Can Computer Think Like A Man". The article, of course, created a flutter but it ultimately established Turing as "Father of Artificial Intelligence". In 1950's, John Backus had given an excellent account of the drawbacks and limitations posed by the existing conventional computational model based on von Neumann concepts in a paper titled "Can Programming be liberated from von Neumann style" []. The research resulted in the development of a computational model based upon logic programming. This class of applicative languages are called functional languages, are increasingly gaining importance in new generation computer models. LISP (List Processing) and PROLOG (Programming in Logic) are two of the functional languages which have found acceptance.

To understand something is to transform it from one representation into another, where this second

representation has been chosen to correspond to a set of available actions that could be performed and where the mapping has been designed so that for each event, an appropriate action will be performed. There is a formal sense in which a language can be defined simply as a set of string without reference to any world being described or task to be performed. Although some of the ideas that have come out of this formal study of languages can be exploited in parts of the understanding process, they are only beginning. To get the overall picture, we need to think of language as a pair (source language, target representation), together with a mapping between elements of each to the other. The target representation will have chosen to be appropriate for the task at the hand.

There are three major factors that contribute to the difficulty of an understanding problem :

- The complexity of the target representation into which the matching is being done.
- The type of mapping : one-one, many-one, one-many, or many-many.
- The level of interaction of the components of the source representation.

Developing programs that understands a natural language is a difficult problem. There is much ambiguity in a natural language. Many words have several meaning and sentences can have different meaning in different context. It requires that a program transform sentences occurring as

part of a dialog into data structures which convey the intended meaning of the sentences to a reasoning program. Now, a programming is structured and ambiguity is shunned while designing a programming language. So, understanding and correcting some of the syntax errors in such languages is less complicated, if not less difficult.

Though it can be safely stated, without any exaggeration, that this field is still in its infancy, and research is going on. We have made a small beginning on our part to do our share.

1.2 Objective and Scope of the project

The objective can be phrased as :

Translating "C" programs for execution on a PROLOG-based system. Because of radically different control and data structures in these two languages, the problems associated with translation process are to be discussed and its performance to be accessed.

As we set about the task of building computer program that understands programming language, one of the first things we shall have to do is to define precisely what the underlying task is and what the target representation should look like. Having done that, it will be much easier to define, at least for that environment, what a sentence (statement) means. In general, this means that the reasoning

program must know a lot about the beliefs and goals of the user, and a great deal of 'C' grammar knowledge.

The proposed work is to develop a software which will be able to execute the programs written in procedural language, a subset of 'C' called **SubC**, on the computer systems whose kernel language are based on logic programming. The efficiency of the output code in PROLOG is not our primary concern; our immediate goal is to demonstrate something that does the job.

It is also proposed to develop an parser based on the input SubC programs as an attempt to include automatic removal of some of the syntax errors. The errors that can be removed are

- 1) Spelling verification and correction.
- 2) Replacement of missing delimiters.
- 3) Non declaration of identifiers.

1.3 Motivation for Project Selection

Since its conception by Alian Colmerauer in early 1970's PROLOG has been gaining in a variety of application areas (e.g.natural language processing, expert systems, database query languages, CAD modellers, etc.). One of the objectives of the Japanese Fifth Generation Computer Project is to develop computer systems whose kernel languages(PROLOG) are based on logic rather than on the conventional imperative languages which have been in general use until now. Since the

logic programming and its application has been intensified. At the same time concern has been voiced over the problem of the large base of existing software which is implemented in imperative languages, and what might happen to it if computer systems with these radically different architectures were to replace existing systems. In order to asses this problem a study is conducted here to implement conventional language on a PROLOG machine.

CHAPTER 2

THEORETIC FOUNDATIONS OF TRANSLATOR

This chapter covers basic material which we will use extensively throughout the rest of this dissertation.

2.1 PROGRAMMING LANGUAGES

In computer programming, a programming language serves as a means of communication between the person with a problem and the computer used to help solve it. An effective programming language enhances both the development and the expression of computer programs. It must bridge the gap between the often unstructured nature of human thought and the precision required for computer execution. A high level language should contain constructs which reflect the terminology and elements used in describing the problem and are independent of the computer used. Such a program solution to a given problem will be easier and more natural if the high level programming language is used.

Advantages of high level languages are as follows :

1. Easier to learn and understand.
2. Naturalness and ease with which an algorithm can be written.
3. Portability or relative machine - independence of languages.
4. Modular and hierarchical description of programming tasks, permitting delegation of tasks and division of labor resulting in greater security and

productivity with a minimum of and effort.

5. Permits better documentation resulting in increased reliability and decreased maintenance.

6. Programs can be easily debugged.

7. Structuredness of a language results in a disciplined use of pointers, data structures, flow-of-control constructs etc, i.e. efficiency of use is increased.

2.2 Notation for Grammars

A programming language consists of a set of programs. A grammar is a formal vehicle for generating these programs. Several relations can be defined on the rules of grammar, and these relations can lead to efficient compilation algorithms for the language associated with that grammar. Consequently, the concept of grammar in the formal language becomes a important one.

The study of grammars constitutes an important subarea of computer science called formal language theory. This area emerged in the mid-1950's as a result of the effort of Noam Chomsky, who gave a mathematical model of a grammar in connection with his study of natural languages. In 1960, the concept of a grammar became important to programmers because the syntax of ALGOL 60 was described by a grammar.

Any means for specifying an infinite language should be finite. One method of specification which satisfies this

requirement uses a generative device called a grammar. A grammar consists of a finite nonempty set of rules or productions which specify the syntax of the languages. Many grammars may generate the same language but impose different structures on the sentences of that language.

Definition :-

A grammar is defined by a 4-tuple $G = (T, N, S, P)$ where

T , set of terminal symbols.

N , set of non-terminal symbols.

S , a distinguished element of N , called the start variable.

P , a finite nonempty set of productions in which each production is of form

$$A \rightarrow B \text{ where } A \in N.$$

The set of all words generated by G is called the language generated by G .

2.2.1. Classification of Grammars

Chomsky classified grammars into four classes by imposing different set of restrictions on the productions:

Type '0' :- Grammar whose rules are unrestricted and is called unrestricted grammar.

Type '1' (Context-Sensitive) :- The grammar contains only productions of the form $A \rightarrow B$, where $|A| \leq |B|$ where $|A|$ denotes the length of A .

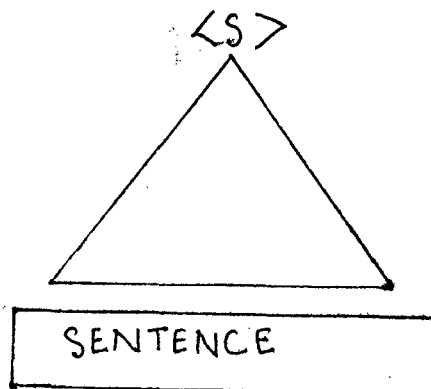
Type '2' (Context-Free) :- The grammar contains only productions of the form $A \rightarrow B$, where $|A| \leq |B|$ and $A \in N$.

Type '3' (Regular Grammar) :- The grammar contains only productions of the form $A \rightarrow B$, where $|A| \leq |B|$ and $A \in N$, and B has the form aS or a , where $a \in T$ and $B \in N$.

Here, we are concerned only with context-free-grammars because they are the most powerful formalisms for which we have effective and efficient parsing algorithms.

2.3 PARSERS

A parser for grammar G is a program that takes as input a string 'w' and produces as output either a parse tree for 'w', if 'w' is a sentence of G , or an error indicating that 'w' is not a sentence of G . Often the parse tree is produced in only a figurative sense; in reality, the parse tree exists only as a sequence of actions made by stepping through the tree construction process. Given a sentence, the construction of a parse tree can be illustrated pictorially in figure below, where root and leaves of the tree are known and the rest of syntax tree must be found.



There are two ways by which this construction can be accomplished.

-- Top-Down Parsing : An attempt is made to construct the tree starting at the root(top) and proceeding downwards towards the leaves(bottom).

-- Bottom-Up Parsing : Completion of the tree is made by attempting to start at the leaves and moving upwards towards the root.

2.3.1 Top-Down Parsing

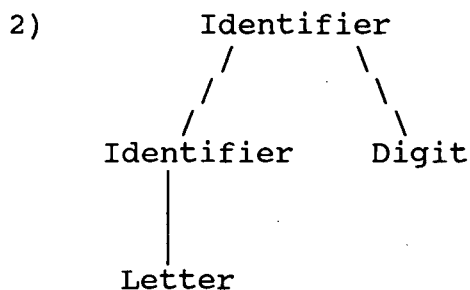
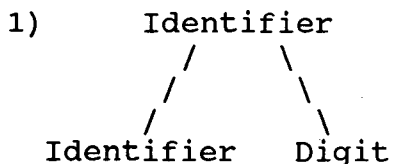
Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently, it can be viewed as attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder. For example, consider the grammar

Identifier ==> Identifier Digit | Letter

Letter ==> A..Z | a..z

Digit ==> 0..9

and the input x4. The construction of the parse tree is as follows :



2) Backtracking :- If we make a sequence of erroneous expansions and subsequently discover a mismatch, we may have to undo the semantic effects of making these erroneous expansions. For example, entries made in the symbol table might have to be removed. Since undoing semantic actions requires a substantial overhead, it is reasonable to consider top-down parsers that do no backtracking.

Removal of backtracking :- In order that no backtracking is required, we must know, given the current input symbol 'a' and the non terminal A to be expanded, which one of the alternates of the production $A \Rightarrow a_1|a_2|a_3|-----|a_N$ is the unique alternate that derives a string beginning with 'a'. That is, by using the next input symbol to guide parsing actions, a proper alternate is detectable. For example,

```
Statement ==>  if Condition then Statement
                | while Condition do Statement
                | do Statement while Condition
```

Then the keywords **if**, **while**, and **do** tell us the unique alternate. One nuance concerns the empty strings. If one alternate for A is ϵ , and none of the alternates is suitable, then we may expand A by $A \Rightarrow \epsilon$.

3) Ambiguous order of alternates :- The order in which the alternates are tried can affect the language accepted. Only way to remove this is to assign the unique alternates at the next input symbol.

4) Error reporting :- When failure is reported, we have very little idea where the error has actually occurred. A top-down parser with backtrack simply returns failure no matter what the error is.

2.3.2 Recursive - Descent Parsing

A parser that uses a set of recursive procedures to recognize its input with no left-recursion and with no backtracking is called a recursive-descent parsing. The recursive procedures can be easy to write and fairly efficient if written in a language that implements procedure calls efficiently.

Left-factoring

Often the grammar one writes down is not suitable for recursive descent parsing, even if there is no left-recursion. For example, if we have the two productions

```
Statement ==>  if Condition then Statement
                |  if Condition then Statement else Statement
```

we could not, on seeing input symbol `if`, tell which to choose to expand statement. A useful method for manipulating grammars into a form suitable for recursive descent parsing is left-factoring, the process of factoring out the common prefixes of alternates.

Let there are two productions

```
A ==> P Q      | P S
```

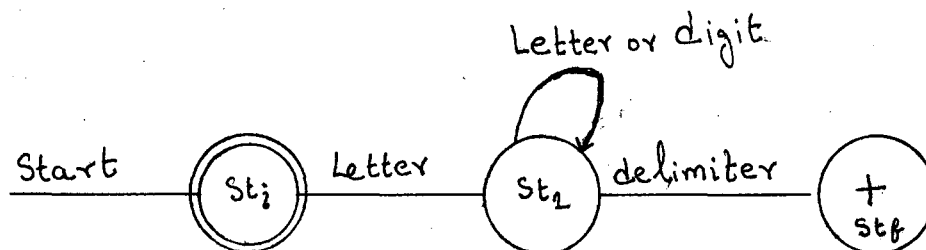
After left factoring, the original productions become

$$A \implies P A'$$

$$A' \implies Q \mid S$$

2.3.3 Transition Diagrams

One way to design any program is to describe the behavior of the program by a flowchart. This approach is particularly useful when the action taken is highly dependent on what token have been seen recently. A special kind of flowchart, called transition diagram has evolved.



Transition diagram for identifiers

In a transition diagram, the boxes of the flow chart are drawn as circle and are called states. The states are connected by various edges leaving a state indicate the input characters that can appear after the state.

2.4 Error Recovery

There are many different general strategies that a parser can employ to recover from a syntactic error. Aho, Ullman, and Sethi [] introduced the following strategies :

1) **Panic Mode Recovery** : On discovering an error, the parser discards input symbol one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon, whose role in the source program is clear. It skips a considerable amount of input without checking it for additional errors.

2) **Phase Level Recovery** : On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue.

3) **Error Productions** : If we have a good idea of common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. If an error production is used by the parser, we can generate appropriate error diagnostics to indicate the erroneous construct that has recognized in the input.

4) **Global Correction** : Given an incorrect input string Ω and grammar G , this method will find a parse tree for a related string β , such that the number of insertions, deletions, and changes of tokens required to transform Ω into β is as small as possible. These methods are too costly, so these techniques are currently only of theoretical interest.

2.5 Syntax Directed Translation Scheme (SDTS)

This method is based on the idea that the type of semantic analysis performed and the nature of the code produced is specified (i.e. determined) for each production of the grammar. Hence, as the production rules for a language are applied, the parser can simply invoke the appropriate semantic analysis and code generation routine and the target code can be produced in a systematic fashion. This type of compiling is often called "syntax directed" because the production rules of the grammar are used to direct the type of processing that is to be performed on the source language statements.

A SDTS is more of a context-free grammar in which a program fragment called an output action (or semantic action or semantic rule) is associated with each production rule. The output action may involve the computation of values for variables belonging to the compiler, the generation of intermediate code, the printing of an error diagnostic, or the placement of some value in a table, for example. The values computed by action $\$$ quite frequently are associated

with a grammar symbol is called a translation of that symbol. The translation may be a structure consisting of fields of various types. If the value of the translation of the nonterminal on the left side of the production is defined as a function of the translations of the nonterminals on the right side. Such a translation is called a **synthesized translation**.

If the translation of a nonterminal on the right hand side of production is defined in terms of a translation of the nonterminal on the left then it is called an **inherited translation**.

CHAPTER 3

IMPERATIVE AND FUNCTIONAL LANGUAGES

The programming languages can be classified into two categories. We will try to understand these two classes which will help in implementing the translator.

3.1 Imperative Languages

Some of the characteristic features are :-

1) In an imperative programming language, the fundamental mode of operation is based on changing the state of variables through assignments or other similar language constructs. These variables are used to imitate the storage of the underlying machine. The basic dependence upon variables and the association of values with variables is characteristic of a von Neumann architecture.

2) The execution of a program which realizes some algorithm may be regarded as a sequence of state transformations in which the state of the store or the current point of control or both may change.

3) The programmer must specify step by step how a result is to be computed.

4) The large syntax base and ever expanding size of the language definition.

5) There is no uniform computational style; each programmer has its own. Style is more of an art than science.

- 6) Declarations which define the attributes of storage location.
- 7) Assignment statements which effect transformations from one state to another by updating storage locations.
- 8) Control statements which determine the point of control at any instant and hence the order of state transformation.
- 9) Structurdness of the language which effect the output on the state at which the procedure or program is called. This demands rigid ordering of procedures.

The conventional languages such as PASCAL, FORTRAN, 'C', etc. are Imperative programming languages. Though new advancements and developments in these languages greatly improved the original versions of the languages, the inherent limitations posed by the von Neumann computational model are not removed and hence the need for new type of programming languages.

3.2 Functional Languages

The second group of languages are called functional languages. LISP and PROLOG are good example in this category. These languages tries to eliminate some of the drawbacks that are existing in the previous class of languages. Some of the features of these languages are :

- 1) They are based on a sound mathematical model which defines semantics of the languages precisely. The semantics of LISP is based on the computational model called λ -calculus and

Prolog is based on a subset of first order predicate calculus formulae known as Horn Clauses.

2) Functional languages emphasizes on what to do on static facts and rules and not on procedural details which involves emphasizes on what to do rather than how to do it exactly; using the specific instruction set provided by the machine.

3) They usually employ some inference mechanism to reach the result like employing a pattern matching technique to access the needed record or the values of the variable.

4) There do not exist variables in the sense of conventional languages. There exists symbols which can be bound or free but they do not represent storage locations. This means that all the values assigned to variables are temporary for instantiation purpose only.

5) The order of evaluation of variables does not change the result and always leads to the same normal form.

6) There is a uniform syntactical notation, which almost completely eliminates the syntax errors and hence the necessity of big manuals for language definition.

7) A important feature of these languages is its typelessness. As far as the computer is concerned it does not distinguish program and data. It is the imperative language which creates an artificial dichotomy due to its structurdness and prevents the program from manipulating it. But in applicative functional languages, since there does not exists any difference between function and data, functions can be used as arguments just as any other data and can be

manipulated accordingly. These meta-level features are very much useful in higher level programming like in the area of Artificial Intelligence.

8) Another distinguishing feature is its dynamic databases. Data can be added or deleted when deemed necessary by programmer. Since functional language do not differentiate between program and data, program parts (clauses) can be added or deleted dynamically. This leads to dynamic programming, which means program can be changed at run time. Learning is an important consequent of this feature.

TH-3928

9) One more striking feature of functional language is its ability to support parallel computation. Most of the problems to be solved will have some parts which can be done concurrently. Conventional languages does not provide ways to express these parallelism. This is basically because the design for development of imperative languages was influenced by the underlying architecture, which is inherently sequential with one-word-at-a-time philosophy.

3.3 PROLOG & 'C'

In order to understand the problems of translation from a conventional imperative language like 'C' to a declarative language like Prolog, we will consider briefly the two languages and difference between them.



3.3.1 The "C" Language

'C' was originally designed for and implemented on the UNIX operating system for the DEC PDP - 11, by Dennis Ritchie. The operating system, the 'C' compiler, and essentially all UNIX applications programs are written in 'C'.

1) **'C' provides a variety of data type.**

The fundamental types are characters, and integers and floating point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures, and unions. Pointers provide for machine-independent address arithmetic.

2) **'C' provides the fundamental control-flow constructions.**

For a well-structured program 'C' provides : decision making (if - else), selecting one of a set of possible cases (switch), looping with termination test at the top (while, for) or at the bottom (do), and early loop exit (break).

3) **'C' is a relatively "low level" language.**

It simply means 'C' deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. 'C' provides no operations to deal directly with composite objects such as character strings, sets, lists, or arrays. There is no operations that manipulate an entire array or string, although structures may be copied as a unit.

- 4) 'C' is not a strongly typed language.
- 5) 'C' provides no input/output facilities.

There is no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called functions.

- 6) Any function may be called recursively.

Function definitions may not be nested but variables may be declared in a block-structured fashion. Variables may be internal to a function, external but only known within a single file, or visible to the entire program.

- 7) 'C' is independent of any particular machine architecture.
- 8) 'C' offers only straightforward, single-thread control flow but not multiprogramming, parallel operations, synchronization, or coroutines.

3.3.2 The Language PROLOG

PROLOG (Programming in Logic) was invented by Alain Colmerauer and his associates at the university of Marseilles during the early 1970's. Prolog is a rule-based language based on the first-order predicate calculus formulae known as Horn clauses. Prolog uses the syntax of predicate logic to perform symbolic, logical computations. Prolog is favored in applications which involves heavy logical deductions throughout.

1) **Prolog is descriptive.**

Instead of a series of steps specifying how the machine must work to solve a problem, a Prolog program consists of description throughout.

2) **Prolog uses rules and facts.**

Facts and rules describe the relationships known to exist between the objects. Problems can be considered in the form of **IF (condition) THEN (action)** rules. This is nothing but the representation of the rule-based organization. There are three form of Horn clauses which forms the basic constructs in Prolog.

a) Prolog Rule.

In Horn clause the rule is of form

$$P \leftarrow Q_1 \cdot Q_2 \cdot Q_3 \cdot \dots \cdot Q_m$$

Here the predicate on the left-hand side is defined to be conjunction of goals on the right-hand side, e.g.

`son(X,Y) :- father(Y,X).`

which may read as 'X is a son of Y if Y is a father of X'.

b) Prolog Facts.

In Horn clause the fact is of form P

Example `son(john,garry)`

which may be read as "john is the son of garry".

c) Prolog Queries .

Given a database of facts and rules, we may ask queries.

It reflects the question to be answered. In Horn clause the form is

<-- Q₁ · Q₂ · Q₃ · - - - · Q_m

Example ? son(john,X)

which may be read as "of whom is john the son ?".

The response to the queries are given by returning the value a variable can take to satisfy the query or simply with yes(true) or no(false).

3) Prolog can make deductions.

Prolog uses backward chaining to deduce facts and the forward chaining can be easily simulated using backward reasoning. The backward reasoning method works backwards from the goal state space. We start with the goal we want to prove, and we try to establish all the facts needed to reach the goal. This reasoning is backward and is called goal directed, top-down, or consequent reasoning.

4) Prolog program execution is controlled automatically.

When a program is executed, the system tries to find all possible sets of values that satisfy the given goal by using backtracking.

5) Prolog uses unification to compute results.

Prolog uses pattern matching mechanism to achieve unification of variables. Thus, it makes easy to manipulate the dynamic database, which Prolog provides. A pattern can be viewed as kind of sketch of an element in the database. Each

pattern is a structure made of different variables. A pattern is said to match a structure if it can be made identical to that structure by replacing its variables by specific values. This process is called pattern matching by unification.

7) Prolog uses backtracking to satisfy subgoals of a goal.

To satisfy goal, the Prolog searches alternative to satisfy subgoals using backtracking. Backtracking is a unique feature of Prolog not found in any other programming language.

8) Prolog is inefficient for numerical processing.

3.3.3 Difference between PROLOG and "C"

Prolog is a relational language, that is a language for logic programming. In contrast, 'C' is a conventional procedural language. In a procedural language, one specifies step by step how a result is to be computed. In Prolog we describe what the relationships are among the entities, rather than how.

The 'C' language have data and program structures such as arrays, records, if-else and loops. There is no such constructs in Prolog. Prolog extensively uses recursion and a unique backtracking mechanics. Prolog variables do not

represent storage locations. All values assigned to variables are temporary and kept only for the duration of a specific execution of the clause. The programmer cannot increment a variable value as, for example, $N = N + 1$ is done in 'C' language. A Prolog procedure is a collection of rules rather than a single closed module of a subroutine.

CHAPTER 4

SYSTEM ANALYSIS AND DESIGN

After stating the objectives of the project in the simple terms, now we go onto the next stage of software development, that is, Analysis. System Analysis is a critical step in developing software systems and programs because it affects all the development step that follows. Analysis is a process of defining the requirements for a solution to a problem. During analysis the needs of the user are examined, and the properties that the system should possess to meet those requirements are identified. The functions to be performed are precisely defined.

System design is a process through which requirements are translated into a representation of software system. Design builds coherent, well planned representations of programs that concentrates on the interrelationships of parts at the higher level and the logical operations involved at the lower level.

In this chapter , I will be describing the subset of language 'C' (SubC) this software can handle. How simple or complex problems can be addressed to, and in what manner. In order to asses the seriousness of a problem, how the study will be conducted and it will be investigated whether that problem is to be translated into Prolog. Also, justification will be given for each of the problem addressed. Constraints and limitations of the software will also be mentioned in this chapter.

4.1 TRANSLATOR

The three main parts of a translator is

1. The source language :- Language which has to be translated into target representation.
2. The target language :- It represents the goal of the problem.
3. Implementation strategy :- Scheme for converting the source language to target language.

4.2 The Source Language

Each programming language has its own unique features. For a specific problem one should choose a language in which the problem can be stated in the most natural way and its solution is easy to assimilate. Since 'C' is most widely accepted programming language nowadays and it captures most of the features of procedural languages, it is chosen as our representative for the source language of the translator. For the sake of simplicity and understanding a subset of 'C', SubC, is considered instead of the full language 'C'.

SubC (Subset of 'C')

As the name implies, SubC is a subset of standard 'C' which includes basic features of 'C' language. SubC is described in detail by its grammar (See appendix A). The

grammar of SubC is a non-ambiguous and context-free. No backtracking is allowed when going from start symbol to terminals (see chapter 2). The grammar of subC is written after removing left recursions in the rules and also, after doing the left factoring to make it suitable for top-down parsers. For clarity and simplicity, SubC is also explained with the help of transition diagrams. For translation diagram of SubC see Appendix B.

Here in short we will explain the features of SubC.

- 1). SubC includes fundamental data type such as integer, floating point and character. In derived structured data types only arrays of simple types are included.
- 2). SubC includes most of control-flow constructs. IF - ELSE, SWITCH, WHILE, FOR, DO - WHILE, and BREAK are all included. No GOTO statements are allowed in 'C'. Since structured programming do not permit goto statements, this omission is not important.
- 3). A printf and scanf statement is included in SubC for input/output. This feature is not there in standard 'C', as printf and scanf are library functions.

As some of the features of 'C' language are not included here, we use the name SubC instead of 'C' throughout the rest of this report.

4.3 The Target Language

PROLOG is chosen as a representative of all functional

languages in our implementation. Prolog being a relational language is becoming more popular and is widely accepted as the language for artificial intelligence applications involving logical and symbolic manipulations. It has gained greater credibility since the effort to develop several different computer systems whose underlying kernel languages will be variants of logic programming, i.e. Prolog, programming in logic. The interpreter for Prolog are easily available and there exist different dialects of it, but all of them have the same basic constructs. We have chosen standard Prolog as written by Clocksin and Mellish as our target code. There is no need to declare data or clauses in this representation.

4.4 The Translation Scheme

The translation technique followed is syntax directed translation scheme. The actual translation process is described in detail under this topic.

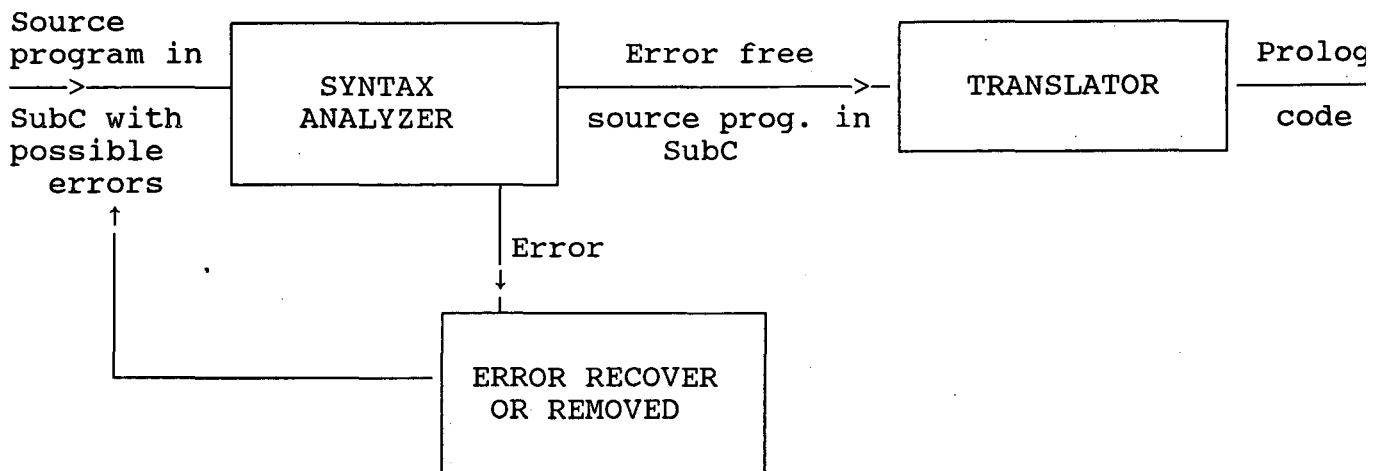
Conceptually, we have divided the whole process in following three phases :

1. Analyzing the source program : This part breaks up the source program into constituent pieces and recognize the source language constructs.
2. Semantic actions : This part executes semantic rules associated with each rule of source language.
3. Getting target program : This deals with the ordering the

resultant segments of the target language into meaningful programs.

Since the efficiency of the output code is not our primary concern, we have ignored the code optimization phase of the translator.

The translation process can be simply viewed in the following figure



4.4.1 Analysis of the source program

The entire process of analysis is divided into two phases :

1) **Linear Analysis** :- In linear analysis, also known popularly as lexical analysis or scanning. The stream of characters making up the source code is read from the top-to-bottom, left-to-right and are grouped into logical units

known as **tokens**. Tokens are sequences of characters having a collective meaning.

Every character encountered is processed in this phase.

White space characters (Blanks, tabs) act as token delimiters and are ignored by the scanner.

A line number counter is incremented whenever a newline character is encountered. This facilitates displaying error messages, giving the correct line number where the error has occurred.

End-of-file character is used for keeping track of file end and ending the program.

Letters and digits invoke different conditions. They are grouped together into a token until a delimiter is encountered. These tokens are divided into two types, Numbers and Identifiers. Numbers can be either of integer type or of floating point type.

All other characters are treated as delimiters and each is processed separately. For example,

if '+' is followed by '=', then "+=" is a single token and its attribute is ASSIGNMENT.

else if '+' is followed by '+', then "++" is a single token and its attribute is INCREMENT.

else '+' is treated as a separate token.

Lexical analyzer also maintains a symbol-table in which every identifier and number is entered along with its

attribute value. Lexical analyzer function returns a integer value, which can be used by the parser to identify the token which is then removed from the input so that processing of the next token can begin when demanded by the parser.

2) **Syntax Analysis** :- It is also called hierarchical analysis. It involves grouping the tokens into grammatical phrases that are used by the program to synthesize output code. The grammatical phrases of the source program are represented by a parse tree.

The hierarchical structure of the program is expressed by recursive rules. For example, we have the following rules as part of the definition of statements.

if exp_1 is an expression and $stmt$ is a statement, then
do { $stmt$ } while exp_1 , and
if (exp_1) $stmt$
are statements.

Similarly, if exp_1 and exp_2 are expressions, then

$exp_1 ++$,
 $exp_1 + exp_2$
are expressions.

We divide the analysis into two parts because it simplified the overall task. One factor which influenced the division was inherently recursive nature of SubC constructs. Fundamentally, lexical constructs do not require recursion, while syntactic constructs of SubC do. Context - free grammar was used as a formalization of recursive rules to guide syntactic analysis. CFG's was introduced in Chapter 2 and

grammar of SubC is given in Appendix A.

In our program, the parser obtains a string of tokens one-by-one from the scanner and verifies whether the string can be generated by the grammar of CFG or not. If it cannot be generated then the parser reports the syntax error. It would also recover from commonly occurring errors so that it can continue processing for the remainder of its input without much hindrance.

We have used predictive parser to recognize the SubC language constructs. By carefully writing the grammar, eliminating left recursion from it and left factoring the resultant grammar, we had obtain a grammar that is parsed by a top-down, recursive descent parser that needs no backtracking. That is, the proper sub-routine is invoked by looking at only the first token it derives. For example, flow of control constructs in SubC are detected in this way.

```
statement ==> if ( expression ) statement
           ==> while ( expression ) statement
           ==> for ( statement_2 )
```

Keywords **if**, **while**, and **for** tells us which alternative rule is the only one that could possibly succeed.

We have taken the help of transition diagrams in implementing predictive parser. As shown in Appendix B, there is one diagram for each non-terminal.

Our predictive parser algorithm based on the transition diagram attempts to match terminal symbol against the input, and makes a potentially recursive procedure call

whenever it has an edge labeled by a non-terminal. A transition on a non-terminal X is a call of the procedure for X . The parser behaves as follows :

- a) It begins in the start state for the start symbol `translator_unit`.
- b) After some actions it is in state α with an edge labeled by terminal w to state β .
if (the next input symbol is w) then
 the input moves one position right and the parser goes to state β .
if (edge is labeled by a nonterminal X) then
 parser invokes the procedure for X , without moving the input cursor.
- c) If it reaches the final state then the input program is accepted, else it is given to error handling routine.

4.4.2 Error Handling Technique

The syntax analysis phase handle a large fraction of the errors detectable by the translator. The lexical phase can detect errors where the character remaining in the input do not form any token of the language. Those errors where the token stream violates the structure rules of the language SubC are detected by the parser. After detecting an error, parser passes the control to the error handling routine, which must somehow deal with that error, so that processing can go on.

Error can be lexical, syntactic, semantic or logical. Our error handling routine can handle only lexical or syntactic type of errors.

Lexical error such as misspelling an identifier or a keyword is handled in a simple way.

a) Handling misspelled keywords

-- All the keywords in SubC is inserted into symbol-table along with their respective attribute values.

-- During the scanning, each identifier or number excluding the keywords are inserted in the symbol table with their respective attribute value.

-- Now if a keyword is misspelled, translator will recognize it as a new identifier and will try to insert it into the table. But afterward the token was found to be disobeying the grammatical rule defining the statement of SubC.

-- This misspelled token is withdrawn from the symbol table and compared with the list of all keywords.

-- Using pattern matching technique, the translator will try to identify the actual keyword which is then replaced in its rightful place.

b) Handling misspelled identifiers

The strategy is same as above with slight modification. Here, all the variables written in the declaration parts are treated as correct one and are entered in symbol-table. Then all the identifiers in the statement part of the programs are treated as in (a). Since, identifiers can differ in single character place, error

handling outline will ask the user to verify the correction before it actually replace it.

Since, incorrect numbers (integer or floating point) are not lexical errors, the translator will treat each number as logically correct.

Error handling routine can recover some of the simple errors and can detect the remaining errors. To, recover the errors, the routine uses the phrase - level recovery (introduced in chapter 2) strategy. On detecting an error, the parser performs local correction like replacing a comma by a semicolon, delete an extraneous semicolon, or insert a missing character.

Example 1 : The correct sentence in SubC is

#include < filename >

where filename is a non terminal.

a) Suppose, the input program has the following incorrect sentence instead of the above sentence

include < filename >

that is, '#' is missing.

Now, as soon as parser find **include**, it detects an error. Since, according to the grammar rule, include should be preceded by '#', the error handing function replaces the missing '#' in the input stream.

b) Input sentence is # < filename >

The parser on encountering '<', will detect an error. It will use a lookahead pointer to go to '>', then it can be safely stated that the error is missing token **include**.

c) Input is `?include < filename >`

Same as in (a) but here the parser decide that '?' is wrongly placed and will replace it with '#'. .

Example 2 : The correct sentence in SubC is

```
int x,y,z; \n
```

where `int` is a keyword and `x,y,` and `z` are identifiers.

In SubC semicolon acts as a statement terminator.

a) Input sentence is `int x,y;z; \n`

According to the grammar rule, the parser will detect the error only after reaching the identifier `z`. On looking ahead it will see the second ';' followed by a newline character. The condition that the whole sentence is written in a single line points to the probable error of first ';' placed erroneously instead of ','.

b) Input is

```
int x,y; \n
z; \n
```

In this case, the parser will decide that a `type_specifier` (`int`, `float` or `char`) is missing.

Since, there is no logical difference in above two sentences, the parser will ask the user to verify the change before actually replacing it in the input stream of tokens. In any case, this strategy is used to successfully recover from such class of errors.

The error handling function uses the strategies of panic mode and phrase level recovery to deal successfully

with lexical and syntax error in input SubC program. The method can be made more intelligent by increasing the range of the lookahead. Our function can lookahead until a newline character is encountered.

4.5 Translation Actions

We have followed a state transformation approach introduced by Williams and Chen [5] for translating SubC into Prolog.

In state transformation approach a block in source language can be regarded as an action which transforms some initial state S_0 into some final state S_n . Each statement T_i ($1 \leq i < n$) of the block will transform state S_{i-1} into state S_i , so that the execution of the block as a whole may be represented by the sequence

$$\{ S_0 \} T_1 \{ S_1 \} T_2 \dots \{ S_{n-1} \} T_n \{ S_n \}$$

in which T_i is the i th statement of the block. If each statement T_i is thought of as being represented by a Prolog rule, a block consisting of statements T_1, T_2, \dots, T_n may be translated as

$$\text{block}(S_0, S_n) \leftarrow t_1(S_0, S_1), t_2(S_1, S_2), \dots, t_n(S_{n-1}, S_n)$$

where the subgoals t_1, t_2, \dots, t_n are the names of predicates defining the effects of statements T_1, T_2, \dots, T_n .

4.5.1 Declarations

Handling declarations of SubC does not create much problem because in Prolog declaring variables are not required. As Prolog variables do not represent storage locations, hence there is no need to declare them explicitly. In our approach the declaration part and statement part are taken as two different parts and are treated separately.

During the declaration part of input SubC program, the parser will take only the book-keeping actions. Information such as the name , type and scope (in case of an array) of each variable is stored in symbol table and the variable table, for use in subsequent parts. This information prevents the undisciplined use of variables. A variable declared under two different data types will be detected as an error by the parser.

So the parser will go through the declaration part of SubC without translating any of it into target representation, Prolog.

4.5.2 Statements

As mentioned before, the statements are parsed in top down fashion with information being retrieved and stored as required.

A) Control Statements

Control statements that are used in SubC are not available in Prolog. We have taken the case of various

control structures one by one.

A.1) if - else statement :

If - else statement is represented as

```
if ( condition ) statement1 else statement2
```

In Prolog, it is represented by replacing the **if** statement with the body of a separate clause. If-else statement of SubC is replaced with a procedure call and a clause is defined, with one rule corresponding to **if** and other rule for **else**.

The Prolog rule generated for an if-else statement is

```
if_else(IN,OUT) :- condition,  
                  statement1.
```

```
if_else(IN,OUT) :- statement2.
```

If the condition is satisfied and found to be true then the goals for the **if** clause will be executed. Otherwise the second rule will be attempted and the goals for the **else** clause executed. If the **else** clause is absent, the following rule will be generated.

```
if_else(IN,OUT) :- condition,  
                  statement1.
```

```
if_else(IN,IN).
```

The above rules state that if the condition fails, then the state vector will not change.

IN and OUT are two state vectors which are empty for the main body of the program. For each procedure or statement body, the variables in set IN will be the variables accessed in the body. The variables in OUT set will be the variables

modified in the body. Set IN includes all variables used in the statement body together with any variable in OUT which return values if the variables are not updated.

A.2) Switch statement :

The body of switch statement is

```
switch ( expression )
{
  case L1 : stmt1; break;
  case L2 : stmt2; break;
  .
  .
  .
  case Ln : stmtn; break;
  default : stmtn+1; break;
}
where L1, L2, ....., Ln are labels.
```

The switch statement is similar to if-else statement, with the number of alternatives more than two.

The Prolog code generated for a switch statement is also similar,

```
switch(L1,IN,OUT) :- stmt1.
switch(L2,IN,OUT) :- stmt2.
:
:
:
switch(Ln,IN,OUT) :- stmtn.
switch(_,IN,OUT) :- stmtn+1.
```

A.3) Iterative Statements (for,while and do) :

The format of the code generated for a while or do or for statements will be similar. Hence, we are considering only the case of while statement. Since, do and for loops can

be proved to be equivalent to **while**, they are expressed in terms of **while** only.

Prolog does not facilitate any loop in an explicit form. One way of simulating such a loop construct is to use recursion. Now, the statement is

```
while ( condition ) statement
```

The Prolog code generated for the **while** statement is

```
while(IN,OUT) :- condition,  
                 statement,  
                 while(IN',OUT).  
  
while(IN,IN).
```

where IN' is the modified bound conditions.

The rules state that to execute a **while** statement, it is necessary to satisfy the bound condition first. If it holds, the goals for the body will be called one by one, the bound condition will be modified and the goals will be called to perform further iterations. If the condition fails, the current state will be returned as the final state by the second rule.

B) Assignment Statement

Each time a variable is updated by an assignment, a new variable is created to substitute for the old one. This is done because there is no storage location concept in Prolog. Variables can be either bound or free. Once a variable is bonded by a value, it can not be changed unless

it is freed. To accomplish this, recursion is used (because every time the rule call itself, it means a new call for the rule).

To create a substitute for an old variable the parser keeps a record of each SubC variable and a counter to keep track of number of times that variable occurs in input program. The counter is incremented whenever the variable is updated by an assignment statement. Then, the name of the variables concatenated with the counter value to create the new variable.

Assignment statements and input-output statement will not be defined as separate rules, instead they will be translated directly in the context of their parent rule.

4.5.3 Common Data Structures

a) **Variables** : Prolog variables are local; that is, their values remain only for the rule only. The variable value are passed from a rule to other rule.

They are passed as procedure parameters, as, for

example `p(A1,A2,....) :- q(A1,A2,....), ...`

`q(A1,A2,....) :- t(A1,A2,....),lm5`

b) **Arrays** : There are no arrays in Prolog. We have chosen Prolog lists as the array. Though this method is inefficient, it was preferred because it is easy to implement.

In SubC, the access an element of an array is random while the Prolog lists are only sequential. All the operations which can be applied to an array are also possible for a list.

Whenever an element at position i of an array is updated, the following operations are done on the corresponding list

- Go to the i th element of the list.
- Remove that element from the list.
- Insert the new value at the i th place in the list.

The above technique was used for developing a prototype source-to-source translator whose test results are given in Appendix C.

CHAPTER 5

CONCLUSION & SUGGESTED ENHANCEMENTS

5.1 CONCLUSION

After finishing the system testing, which includes the last review of the objectives, we look back to analyze the goal we had set at the start. What we were able to achieve and what we fail to cherish.

Our primary objective was to develop a source-to-source translator which translates the program written in 'C' into Prolog rules. We have taken a subset of 'C' language, SubC, for the sake of simplicity and understanding the difficulties involved in translating radically different data structures of these two languages. One of the objectives of the project was to provide the overview of the central themes, ideas, and difficulties associated with error recoveries. The core idea for translating SubC into Prolog was to use syntax directed translation scheme.

Of the above objectives stated, we were able to attain most part of it and developed a prototype translator that translates SubC into Prolog. The syntax errors where the token stream violates the structure rules of the language SubC were detected and was effectively dealt with by the predictive parser of the translator.

In all it was a very useful and fruitful experience during which we gained insight into this fascinating field of source-to-source translation of programming languages which is so much related to language understanding and particularly to programming languages.

5.2 SUGGESTED ENHANCEMENTS

Though we tried our best to make the software as broad as possible in limited time, there is still a wide range of enhancements, which can be instituted into the system to make it more flexible and hence improve its quality.

Following are the some of the enhancements suggested by us

- 1) Range of SubC can be increased by including record and file structure.
- 2) Another omission which SubC makes is the function call. Since the standard library functions can be written in Prolog these function calls can be incorporated by writing, beforehand, the equivalent clauses in Prolog.
- 3) Dynamic data types like pointers can be implemented. A pointer variable which points to a data object can be simulated by creating a symbol and binding this symbol to the object to which it is supposed to point.
- 4) The array in SubC is equated to list in Prolog, which is inefficient. A better solution is binary tree representation. This can be implemented as the enhancement to original representation.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Haynes, C.
Logic Continuations.
proc of third international conference on Logic Programming. London, Lecture Notes in Comp. Science, 225, p 671, 1980.
2. Henderson, P.
Functional Programming, Application and Implementation.
Prentice Hall, 1980.
3. Hughes, R. J. M.
The Design And Implementation Of Programming languages.
Ph.D. Thesis programming research group, Oxford, Sept 1986
4. Backus, J.
Can programming be liberated from von Nueman style ?
Comm. of Assoc. of Computing Mach. 21, p 615.
5. Williams, M.H. and Chen, G.
Translating Pascal for execution on Prolog-based systems
The Computer Journal, 29, 3, p 246, 1986.
6. Munakata, T.
Procedurally Oriented Programming Technique in Prolog.
IEEE Expert, 1986.

7. Subramanian, V.S. & Bharadwaj, K.K. & Sharma, A.K.
Automatic Programming : Transforming Recursive
Specifications into Logic Program.
Computer Science and Informatics, 15, 1.
8. Berghal and Traudt.
Spelling verification in Prolog.
ACM SIGPLAN NOTICES, Vol. 21, No. 1-5, 1986.
9. Aho, A.V. & Sethi, Ravi & Ullman, J.D.
Compilers. Principles, Techniques and Tools.
Addison - Wesley, 1986.
10. Leinius, R.P.
Error Detection and Recovery for Syntax Directed
Compiler Systems.
Ph.D. Thesis, University of Wisconsin, Madison.
11. Kernighan, B.W. and Ritchie, D.M.
The 'C' Programming Language.
Prentice Hall, 1977.
12. Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Norosa Publishing House.

APPENDICES

APPENDIX A

GRAMMAR OF SUBC

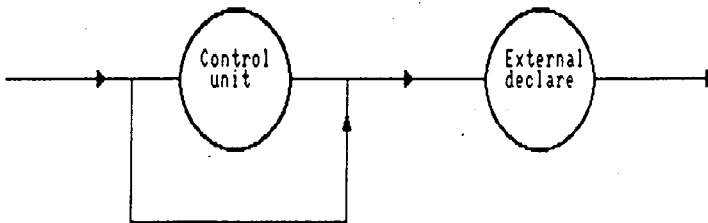
translator_unit	====>	control_unit	external_declare
			external_declare
control_unit	====>	#	control_unit_1
control_unit_1	====>	include	< identifier > control_unit
			define identifier constant
			control_unit
external_declare	====>	declaration_list	function_def
		compound_statement	external_declare
function_def	====>	function_specifier	declarator
			main()
function_specifier	====>	void	char int float
compound_statement	====>	{	compound_stat_1
compound_stat_1	====>	declaration_list	compound_stat_2
compound_stat_2	====>	statement_list	}
declarator	====>	identifier	(declarator_1
declarator_1	====>	type_specifier	id_expression
		declarator_2	

		ϵ
statement_2	====>	expression ; expression ; expression ;
expression	====>	id_expression assignment conditional_exp
conditional_exp	====>	and_exp logical_and_exp
logical_and_exp	====>	and_exp logical_and_exp ϵ
and_exp	====>	equal_exp logical_equal_exp
logical_equal_exp	====>	&& equal_exp logical_equal_exp ϵ
equal_exp	====>	relate_exp logical_relate_exp
logical_relate_exp	====>	== relate_exp logical_relate_exp != relate_exp logical_relate_exp ϵ
relate_exp	====>	add_exp logical_add_exp
logical_add_exp	====>	< add_exp logical_add_exp <= add_exp logical_add_exp > add_exp logical_add_exp >= add_exp logical_add_exp ϵ
add_exp	====>	term moreterm

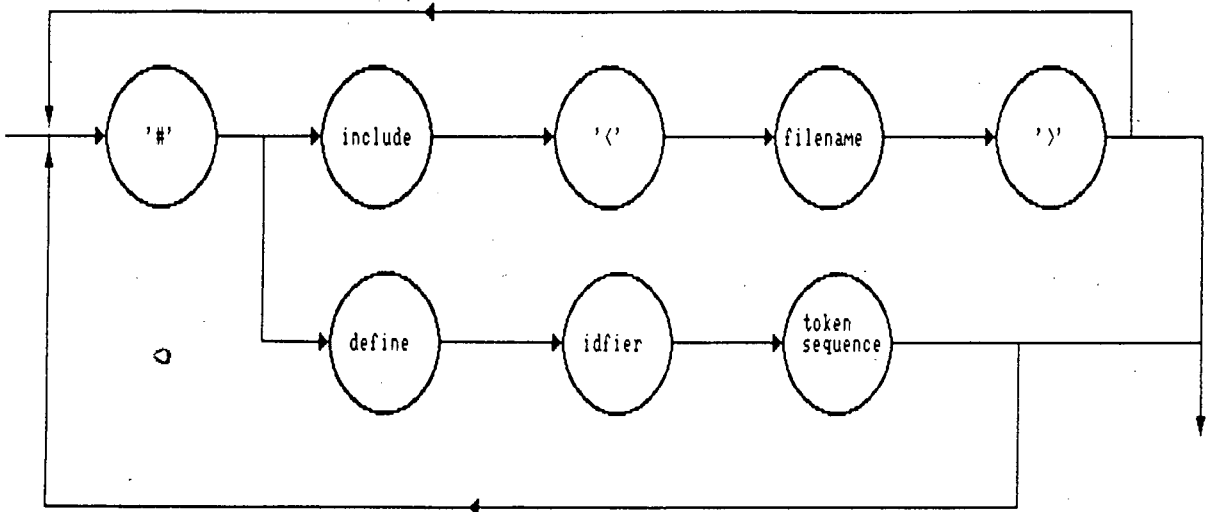
APPENDIX B

TRANSITION DIAGRAM FOR SUBC

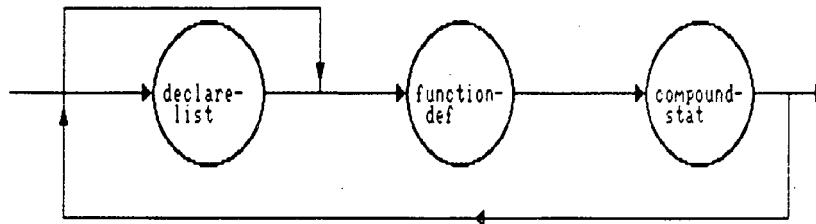
Translation_Unit



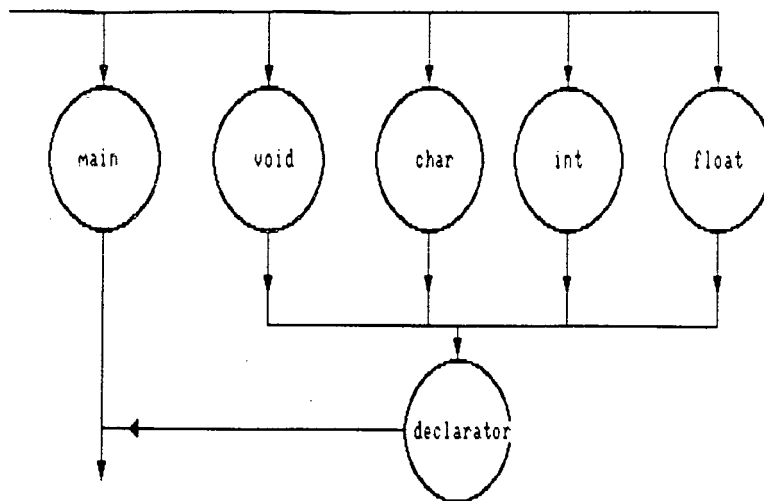
Control_Unit



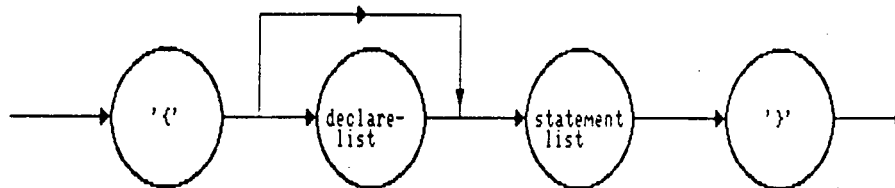
External_declaration



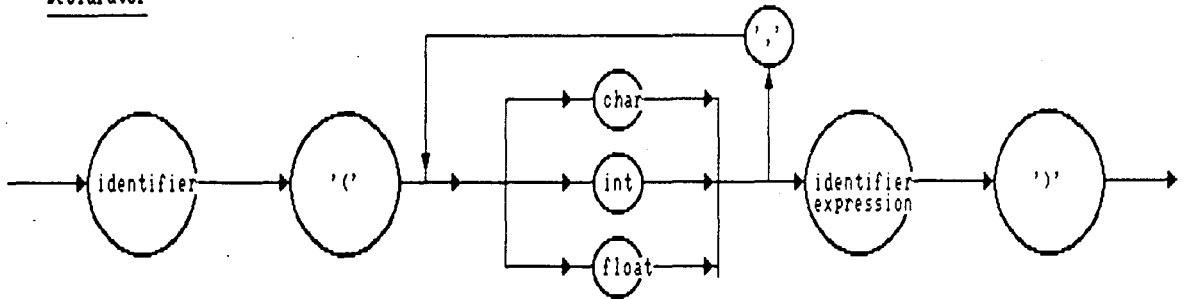
Function_definition



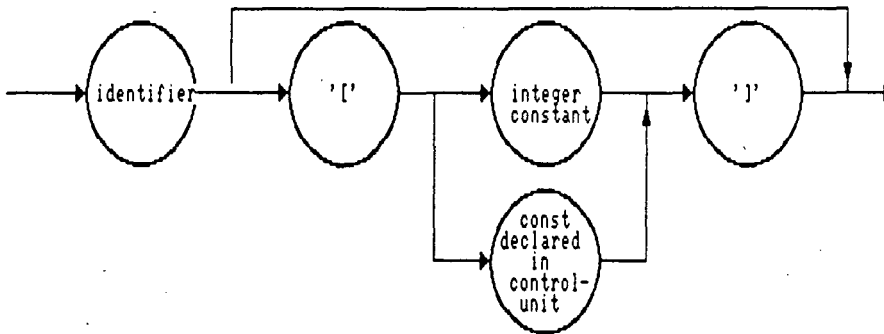
Compound_statement



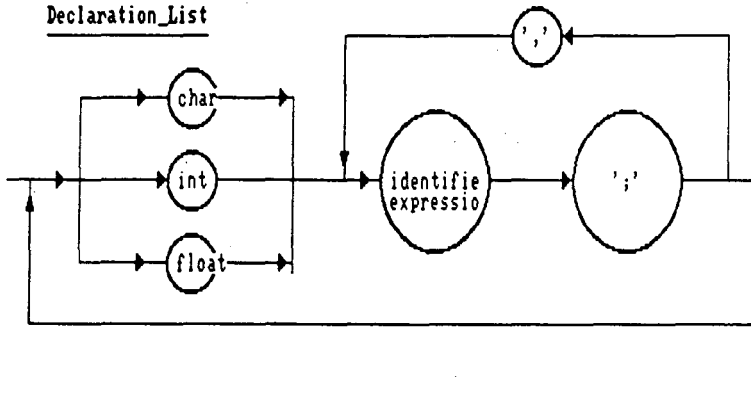
Declarator



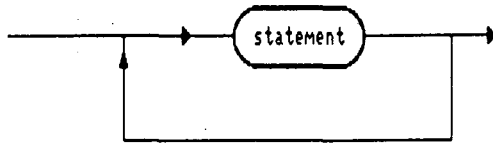
Identifier_Expression



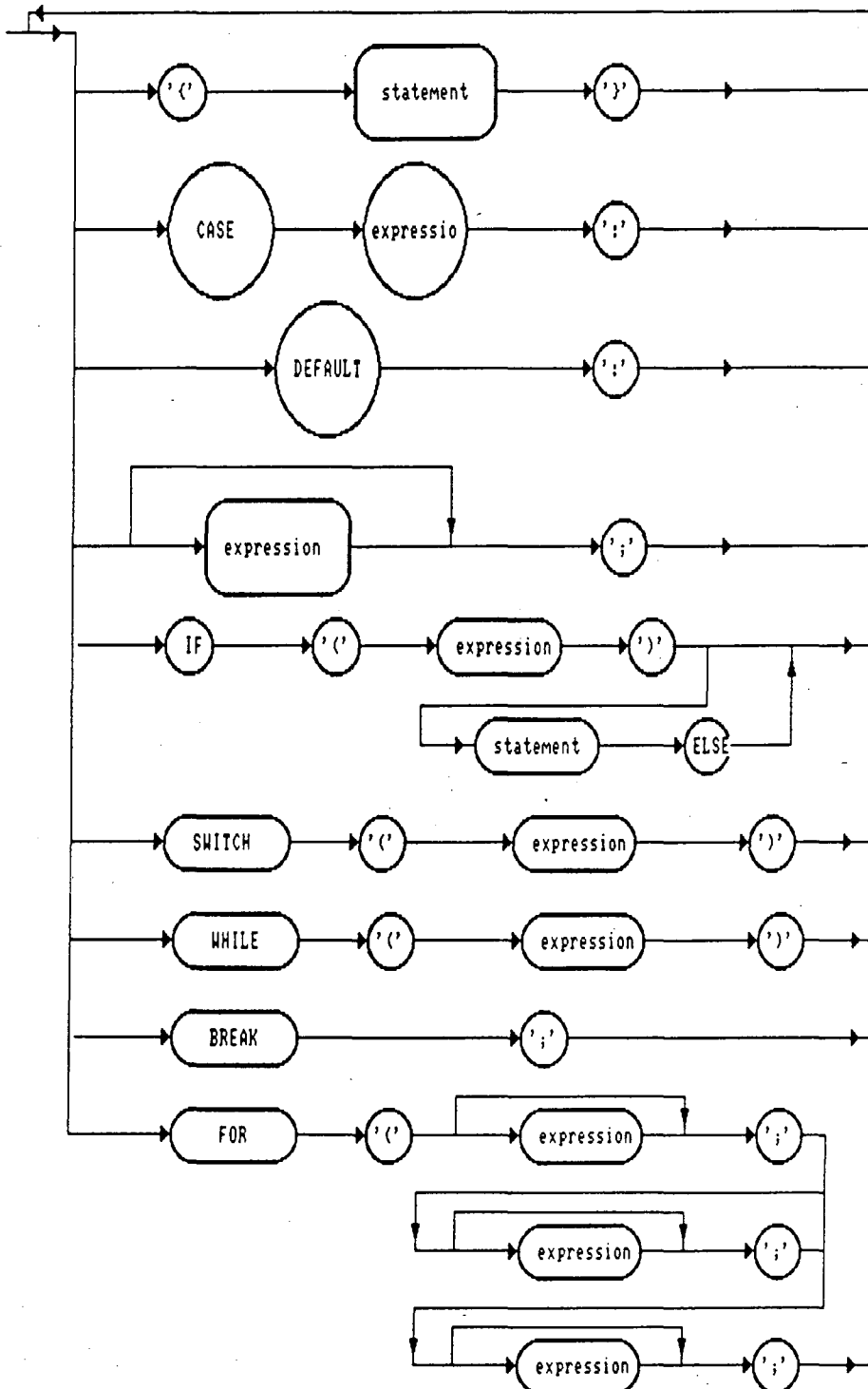
Declaration_List



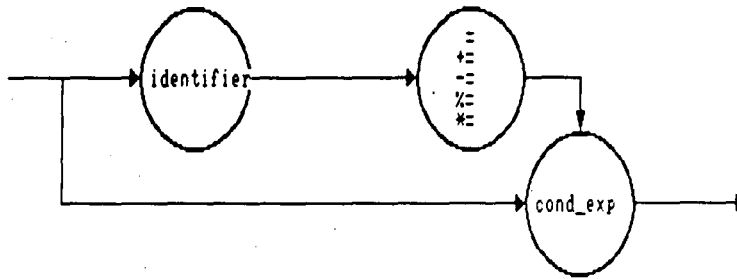
Statement_List



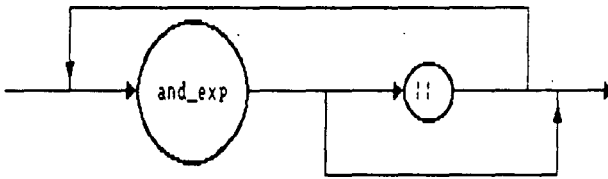
Statement



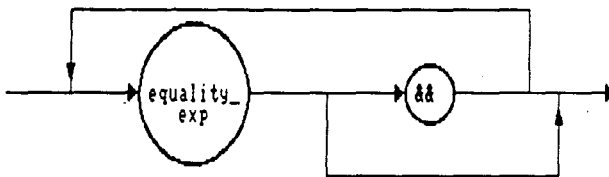
Expression



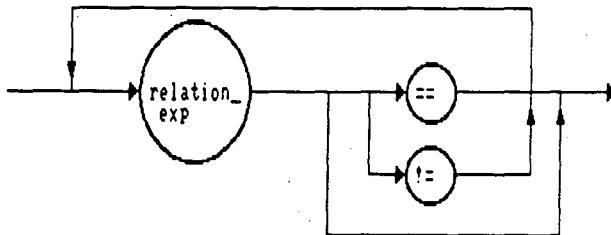
Cond_Exp



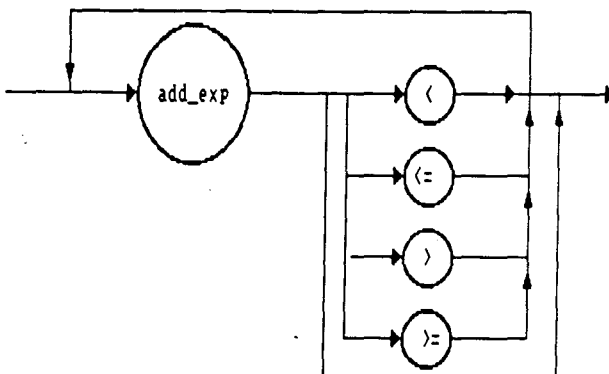
And_Exp



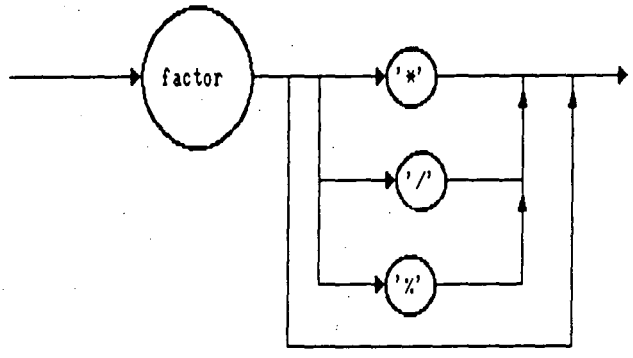
Equality_Exp



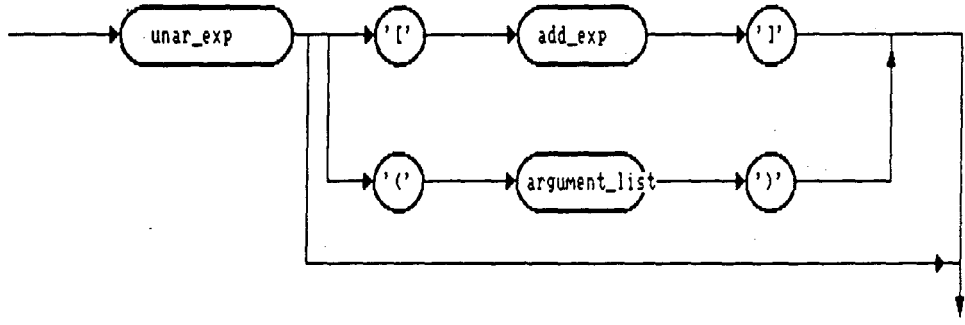
Relational_exp



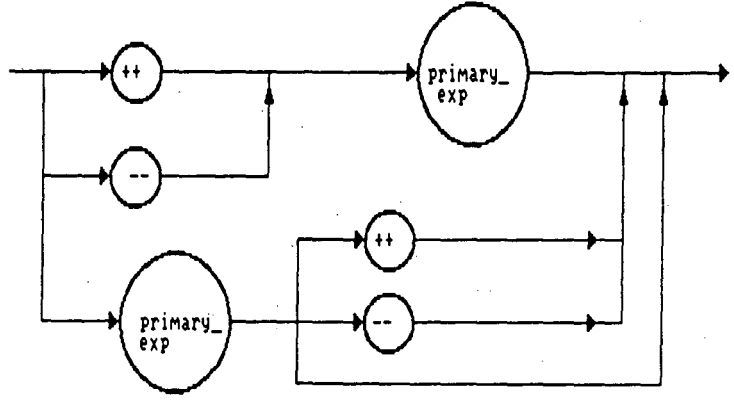
Term



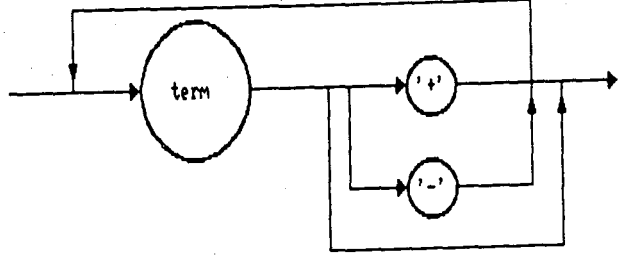
Factor



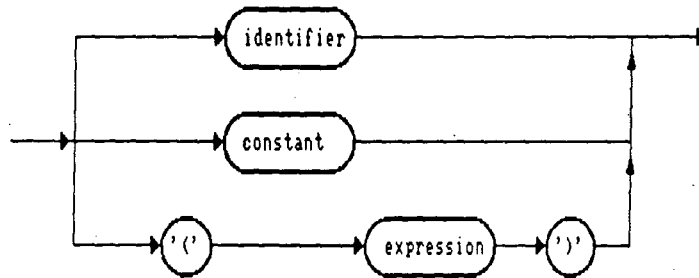
Unar_exp



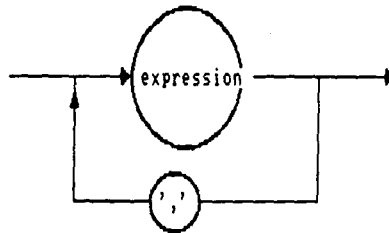
Add_Exp



Primary_exp



Argument_list



File_name : Identifier

Token_sequence : Constant

APPENDIX C

TEST EXAMPLE RESULTS

```
/* Test Example 1. */  
/*This program does nothing. It is used for demonstration  
purpose only */
```

```
main()  
{  
    int a,b;  
    a = 4;  
  
    if (a>0)  
    {  
        a = 9;  
        b = 6;  
    }  
    else  
    {  
        a = 3;  
        b = 6;  
    }  
}
```

```
/* End of Example */
```

```
/* The translated program for test 1 example */
```

```
main :-  
    A1 = 4,  
    if_1(A1,B1,A2,B2).  
  
if_1(A1,B1,A2,B2) :-  
    A1 > 0,  
    A2 = 5,  
    B2 = 6.  
  
if_1(A1,B1,A2,B2) :-  
    A2 = 3,  
    B2 = 2.
```

```

/* Test Example 2 */
/* For If-then-else */

main()
{
    int i,a,b;
    scanf(a,b);
    i = 0;
    if ((a==0) && (b!=10))
    {
        a = a + 2;
        b = a - 4;
        i++;
    }
    else
    {
        a = a + b + 2;
        b += 4;
    }
}

/* End Example 2 */

```

```

/* The translated program for test 2 example */

```

```

main :-
    readint(A1,B1),
    I1 = 0,
    If_1(A1,B1,I1,A2,B2,I2).

If_1(A1,B1,I1,A2,B2,I2) :-
    A1 = 0,
    B1 <> 10,
    A2 = A1 + 2,
    B2 = A2 - 4,
    I2 = I1 + 1.

If_1(A1,B1,I1,A2,B2,I1) :-
    A2 = A1 + B1 + 2,
    B2 = B1 + 4.

```

```
/* Test 3 program.*/
```

```
/* This program finds the max, min and average values of a  
group of integer */
```

```
main()  
{  
    int max,min,n,t,a;  
    float ave;  
    scanf(a);  
    max = a;  
    min = a;  
    n = 0;  
    t = 0;  
    while (a > 0)  
    {  
        if (a > max)  
            max = a;  
        else if (a < min)  
            min = a;  
        n = n + 1;  
        t += a;  
        scanf(a);  
    }  
    ave = t % n;  
    printf(max,min,ave);  
}
```

```
/* The translated program is as follows */
```

```
main :-  
    readint(A1),  
    Max1 = A1,  
    Min1 = A1,  
    N1 = 0,  
    T1 = 0,  
    while_1(A1,Max1,Min1,T1,N1,Max3,Min3,T3,N3),  
    Ave1 = T3 % N3,  
    writeint(T3,N3),  
    writereal(Ave1).  
  
while_1(A1,Max1,Min1,T1,N1,Max3,Min3,T3,N3) :-  
    A1 > 0,  
    if_1(A1,Max1,Min1,Max2,Min2),  
    N2 = N1 + 1,  
    T2 = T1 + A1,  
    readint (A2),  
    while_1(A2,Max2,Min2,T2,N2,Max3,Min3,T3,N3).  
  
while_1(A1,Max1,Min1,T1,N1,Max1,Min1,T1,N1).
```

```
If_1(A1,Max1,Min1,Max2,Min1) :-  
    A1 > Max1,  
    Max2 = A1.  
If_1(A1,Max1,Min1,Max1,Min2) :-  
    if_2(A1,Min1,Min2).  
  
if_2(A1,Min1,Min2):-  
    A1 < Min1,  
    Min2 = A.  
  
if_2(A1,Min1,Min1).
```

