DETAILED DESIGN AND PARTIAL IMPLEMENTATION OF A PRE-PROCESSOR FOR PROLOG PROGRAMS WITH EMBEDDED C STATEMENTS

By

LUKAS B. SANTOSO

Sarjana Teknik Degree

Bandung Institute of Technology

Bandung, Indonesia

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1991

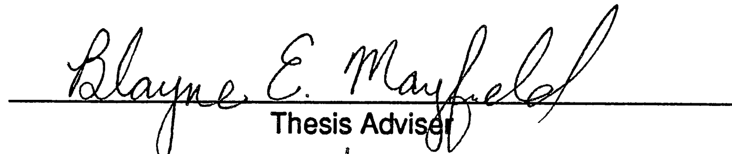
Thesis
1991
S237d
cop.2

DETAILED DESIGN AND PARTIAL IMPLEMENTATION

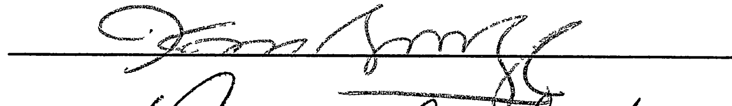
OF A PRE-PROCESSOR FOR PROLOG PROGRAMS

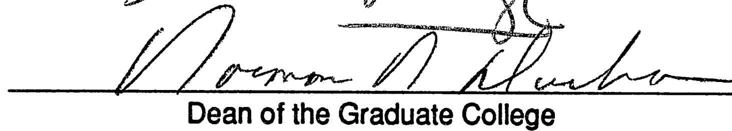
WITH EMBEDDED C STATEMENTS

Thesis Approved:



Thesis Adviser





Dean of the Graduate College

1398529

ACKNOWLEDGEMENTS

I extend my appreciations and thanks to many people without whom this thesis would have never existed.

Drs. Blayne E. Mayfield and Mansur H. Samadzadeh, my thesis advisers, provided invaluable guidance, advice, critique, and support since the early stage of the work. Dr. K.M. George, my advisory committee member, gave useful advice on compiler writing. Dr. George Hedrick allocated his valuable time for reviewing the report.

Dr. Hal Berghel, Director of the CAIES at the University of Arkansas, contributed some useful related materials.

Brendan Machado and Dr. Chang-Hyun Jo gave me motivations with some informal discussions over compiler writing.

Most of the references I needed have been provided in time by the Interlibrary Loan Department of Edmond Low Library, Oklahoma State University.

I also would like to thank Mr. Francis Sunaryo, M.D., for his generous support, encouragement and hospitality during my graduate study. Special thanks are due to my parents, to whom this thesis is dedicated, for their love, patience, and for letting me choose my own career.

TABLE OF CONTENTS

Chapter			Page
I.	INTRODUCTION		1
	1.1	Prolog	1
		1.1.1 Background	1
		1.1.2 Syntax	2
		1.1.3 Unification	4
		1.1.4 Execution	5
		1.1.5 Limitations	7
	1.2	Objectives	11
II.	OVERVIEW OF THE SYSTEM		12
	2.1	Declarative Meaning of Statements in a Program	12
	2.2	The Pre-Processor's Components	17
	2.3	Input and Output	19
	2.4	Predefined Database	23
III.	TOKENIZER AND CLAUSE GENERATOR		26
	3.1	Tokenizer	26
	3.2	Clause Generator	29
IV.	EMBEDDED C RECOGNIZER		30
	4.1	Context Free Grammars and Left Recursion in Prolog	30
	4.2	A Subset of C Syntax	32
	4.3	Top-Down Parser and Left Recursion	32
	4.4	Left-Corner Parsing Method for ECR	33

Chapter Page

4.5. Left-Corner Parsing Implementation in Prolog 38
4.6. Static Semantic Analysis 41
4.7. C-to-Prolog Translation 42

V. SUMMARY AND SUGGESTED FUTURE RESEARCH 44

REFERENCES 46

APPENDIXES 51

APPENDIX A - GLOSSARY 52

APPENDIX B - PREDEFINED DATABASE 54

APPENDIX C - A SUBSET OF QUINTUS PROLOG GRAMMAR RULES 59

APPENDIX D - PARTIAL LISTING OF THE SOURCE CODE OF ECR 64

LIST OF FIGURES

Figure		Page
1.	A Prolog Execution Flow	7
2.	Factorial Program Using 'while' Loop	10
3.	Factorial Program Using 'for' Loop	10
4.	Factorial Program in Prolog	10
5.	if_then_else/3	10
6.	The Pre-Processor Block Diagram	19
7.	The Transformation Process	21
8.	A Prolog Program with Embedded C Code	22
9.	The Result of Transforming the Program in Figure 8	22
10.	Tokenizer	28
11.	Clause Generator	29
12.	The Parse Tree for the String *bbaaab*	36
13.	The Sequence of Configurations for Parsing the String *bbaaab*	37

CHAPTER I

INTRODUCTION

1.1 Prolog

1.1.1 Background

Prolog, which stands for *Programmation en Logique*, was originally created by Alain Colmerauer and his group at the Faculty of Sciences at Luminy in Marseilles, France, in the early 1970's. The original objective of the creators was to integrate Robinson's resolution principle into a programming language [Colmerauer85]. Its initial design goal was to help natural language processing. Since then, many versions of Prolog have been written; e.g., CProlog, Quintus, Arity, Turbo Prolog, Prolog-2, and UNSW Prolog [Malpas87]. More attention was given to this language after the Japanese Ministry of International Trade and Development officially launched their fifth generation project that was to be based on the logic programming software technology in 1981.

Prolog has shown its usefulness in many applications, especially in the areas of nonnumeric processing such as: compiler writing [Warren80], logic circuit simulation [Uehara83], natural language processing [Maruyama84], and expert systems and pattern-directed systems [Bratko86]. Applications involving pattern matching, backtracking, or incomplete information are easily

handled by the language.

Prolog, unlike many other languages, encourages a programmer to think declaratively rather than imperatively, despite the fact that it is possible to use both methods. Declarative thinking refers to 'what' a computer must do to perform a task. On the other hand, imperative or procedural thinking refers to 'how' a computer must do the job. The following illustration shows how Prolog sentence can be read both declaratively and procedurally:

Prolog sentence	:	parent(andy,amy).
Declarative reading	:	'andy' is a parent of 'amy'.
Procedural reading	:	the goal of finding that 'andy' is a parent is satisfied by 'amy'.

Prolog sentence	:	predecessor(X,Z) :- parent(X,Z).
Declarative reading	:	Any X is a predecessor of Z if X is a parent of Z.
Procedural reading	:	To find whether X is a predecessor of Z, find whether X is a parent of Z.

1.1.2 Syntax

A Prolog program consists of a sequence of sentences. Each sentence is a Prolog term. Terms are written as sequences of tokens. Tokens are sequences of characters, which are treated as separate symbols. For interpretation as a sentence, each list of tokens must be terminated by a full-stop token (a period followed by a layout character, such as space or newline).

The syntax of Prolog used in this thesis is Quintus Prolog syntax [Quintus87]. A subset of the syntax rules is shown in Appendix B. A rule such as

```
sentence    -->  clause
                 | directive
```

is read as "a Prolog sentence may be of the form either a clause or a directive". In this rule, sentence, clause, and directive are non-terminals. A non-terminal symbol is also called a grammar category or a syntactic category. Terminal symbols in the grammar rules are printed in the form of bold letters.

Three dots following a non-terminal, such as in digit... (see Appendix B), denote a sequence of one or more non-terminals. A symbol '?' preceding a non-terminal, such as in ?alpha indicates that the non-terminal is optional. Thus, ?alpha... denotes zero, or one, or more non-terminal symbol 'alpha'.

The syntax rules can be divided into three groups: syntax of sentences as terms, syntax of terms as tokens, and syntax of tokens as character strings. Depending on the group, a non-terminal may represent a class of either terms, token lists, or character strings.

Some restrictions on the syntax are:

1. A functor has the form *f(t1,t2, ..., tn)* where *f* is the name of the functor with arguments *t1, t2, ..., tn*. In a term, a name of a functor and its following '(' must not be separated by spaces, newlines, or other characters. For example,

```
father    (X,Y)
```

is syntactically invalid.

2. "-5" denotes an integer, whereas "-(5)" denotes a compound term of which the functor is '-'/1.

More discussion on the syntax is given in Section 4.1.

3. The arguments of a compound term must have precedence numbers less than 1000. A precedence number is used to disambiguate expressions where the syntax of the terms is not made explicit through the use of brackets [Clocksin81]. As an example,

since the precedence of the infix operator ':-' is 1200, it is necessary to write the expression "A :- B" in parentheses as in

retract((A :- B)).

The use of parentheses reduces the precedence of "A :- B" to 0.

1.1.3 Unification

Unification is the important process of matching in Prolog. The purpose of unification is to find a most general unifier (mgu) of two or more terms. By definition, a term is constant, a variable, or a function $f(t_1, ..., t_n)$ where $t_1, ..., t_n$ are terms and $n > 0$. Two terms are identical if there exists a unifier for those terms. A unifier is a set of variable-for-term substitutions. Let sigma be a unifier, then:

$$sigma = \{ x_1 <\!-\!- t_1, ..., x_n <\!-\!- t_n \}$$

where: $x_1, ... x_n$ are variables

$t_1, ..., t_n$ are terms and $n > 0$.

Given two terms P(x) and P(y), for example, a possible unifier is sigma = { x <-- a, y <-- a }. Implementing sigma to both terms, we have:

sigma(P(x)) = P(a) by substituting a for x
sigma(P(y)) = P(a) by substituting a for y

A most general unifier or mgu of two terms is a unifier such that the associated common instance is most general [Sterling86]. In the above example, P(a) is the common instance of P(x) and P(y). The most general unifier for P(x) and P(y) is sigma.

In Prolog, a goal can be unified with the head of the clause if: they have the same name and the same number of arguments, and all arguments can be unified. The rules regulating the unification of

arguments are [Malpas87]:

1. Two constants unify with one another if they are identical.
2. A variable unifies with a constant or a structure. As a result of the unification, the variable is instantiated or bounded to the constant or structure.
3. A variable unifies with another variable. As a result, they become the same variable.
4. A structure unifies with another structure if they have the same name, the same number of arguments, and if all of the arguments can be unified.

1.1.4 Execution

A Prolog program is composed of a set of facts and a set of rules. The set of facts can be viewed as knowledge about a certain subject that is unconditionally true. The set of rules is a collection of statements from which conclusions can be derived whenever certain patterns of conditions are satisfied. Prolog allows a programmer to design an algorithm in terms of logic plus control [Kowalski79]. The logic component, which is represented by both facts and rules, specifies the meaning of the algorithm. The control component determines how a particular problem-solving strategy is imposed. The control component refers to the order of goals in the body of rules and the order of the rules. The following Prolog program serves as an example.

```
parent(andy,amy).                 % fact 1
parent(amy,calvin).               % fact 2
predecessor(X,Z)  :-  parent(X,Z).          % rule 1
predecessor(X,Z)  :-
      parent(X,Y),  predecessor(Y,Z).       % rule 2
```

The symbol '%' indicates that characters following it, on the line on which the symbol appears, are treated as a comment. Comments will not be processed by a Prolog compiler. The pair of predecessor rules form a procedure of predecessor. A procedure is a collection of rules with the same predicate name and the same number of arguments in the head of the rules [Sterling86].

The facts of the parent relationship and the rules of predecessor determine the logical component of the algorithm. The control component is defined by the fact that Prolog uses a top-down and left-to-right approach in examining both rules and facts. Given a query such as "?- predecessor(X,calvin).", Prolog will try to answer it by invoking the following steps:

- Prolog searches a clause in which its head has 'predecessor' as the predicate name with two arguments. The search always begins from the top of the database. The search leads Prolog to use rule 1 first rather than rule 2 because rule 1 is located on top of rule 2 in the database. The head of rule 1 is then unified with the goal of the query. The unification of the query and the head of rule 1 gives the result: X = X and Z = calvin. The symbol '=' indicates that both literals on its left and right are identical.
- Next, Prolog tries to satisfy *parent(X,calvin)* as the only goal in the body of rule 1. If there are more than one goal in the body, Prolog will try to satisfy all of them beginning from the left-most goal. Using the same mechanism, searching from top to bottom and unification, Prolog finds that fact 2 can satisfy the goal *parent(X,calvin)*. Thus, rule 1 succeeds and so does the goal of the query. The answer given by Prolog will be:

X = amy

yes.

Rule 2 is executed only if rule 1 fails to satisfy a query. As an example, suppose the query "?- predecessor(andy,calvin)." is given. Figure 1 shows Prolog execution to answer the query.

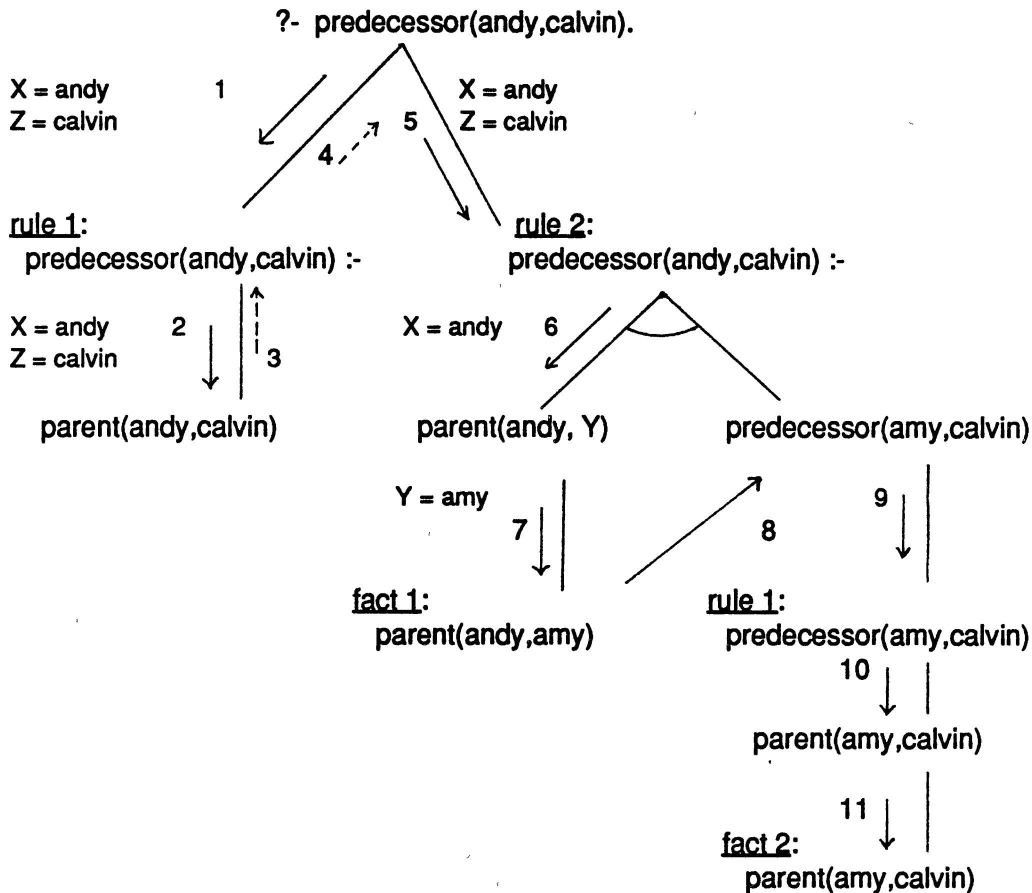


Figure 1. A Prolog execution flow.

The flow of the execution is indicated by the arrow symbols —> and --->. Backtracking is indicated by -->.

1.1.5 Limitations

Although Prolog has shown its strengths in many applications, it also introduces its limitations. Programmers who are already familiar with procedural languages will find that Prolog lacks

certain data structures such as arrays and pointers which are common in procedural languages. Traditional control structures such as while-loops, for-loops, and if-then-else are not explicitly available. Figure 2 and 3 show a program for calculating factorial of N is written in a pseudo-procedural language. Figure 4 shows the program written in Prolog [Sterling86]. This program consists of three types of clauses: initialization, iterative, and result. An iterative clause is a clause which has only one recursive call in its body and zero or more calls to Prolog system predicates before the recursive call [Sterling86]. A user may define the if-then-else control structure as the *if_then_else/3* relationship (Figure 5). '/3' indicates that the predicate *if_then_else* has three arguments. Declaratively, it means that the relation *if_then_else/3* is true if P and Q are true, or (not P) and R are true. An imperative interpretation of this rule is "if P is true then prove Q; otherwise, prove R". The use of cut, '!', in the first clause of *if_then_else* is to prevent backtracking when Prolog fails to prove Q. Thus, the first clause fails when Q fails and this condition leads Prolog to prove R in the second clause instead.

The inherent limitations and the different way of thinking about an algorithm can create a difficult situation even for experienced programmers when they start learning Prolog.

A common attempt to overcome these problems is to link Prolog with procedural languages. Some Prolog versions that exist today have the capability to communicate with one or more procedural languages. Two examples are Quintus Prolog and Arity Prolog. This multilingual capability provides users with the advantage of a combination of features of several languages. One

example of the practical application of this kind of mixed language system is REF (Resource Estimation Facility) which is a bilingual meta-level inference system [Roach90]. In this system, procedures of each language process only the tasks that are suitable for them. For example, I/O functions are done by procedures written in C and Prolog procedures might process the inference mechanism.

A combination of two or more different types of languages does not always guarantee ease of use. Proper data transfer, type conversions, and declarations of modules are some of the points that should be carefully considered. Furthermore, the environment in which the system would be implemented might not always be capable to support all of the languages.

Another attempt to eliminate some limitations of Prolog is by improving it at the unification level [Colmerauer90], i.e., by integrating:

- a refined manipulation of trees, including infinite trees, together with a specific treatment of lists;
- a complete treatment of two-valued Boolean algebra;
- a treatment of the operations of addition, substraction, and multiplication by a constant and of the relation $\neq$.

These new features have been incorporated in a new programming language, Prolog III. The advantage of these new features are not widely known, despite the fact that the prototype of Prolog III interpreter has been in use since 1987 [Colmerauer90].

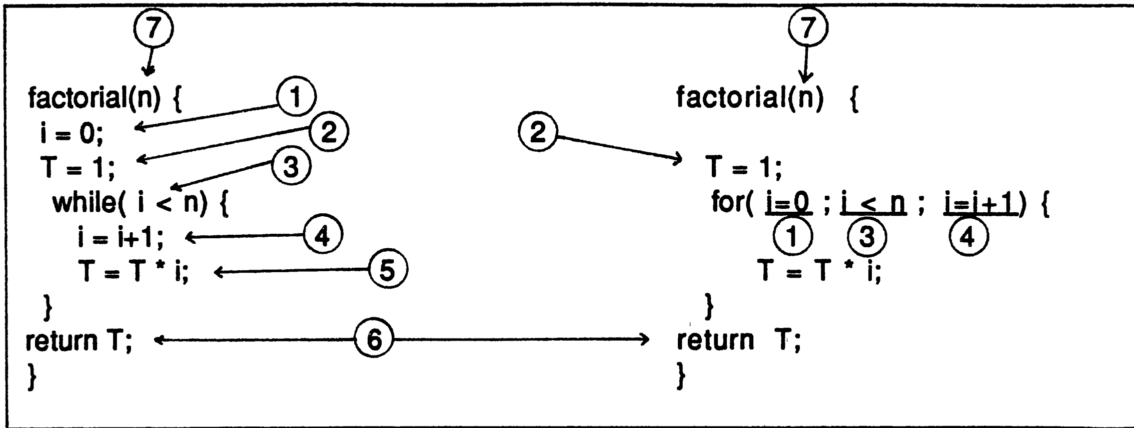
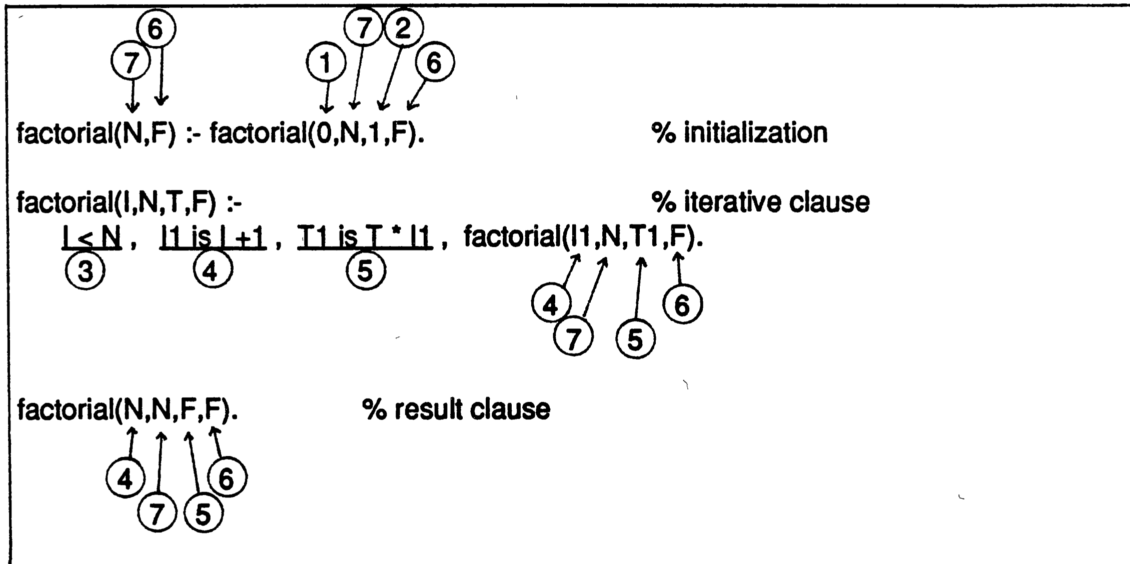


Figure 2. Factorial program using 'while' loop.

Figure 3. Factorialprogram using 'for' loop.



nomenclature:

1, 2: accumulator initialization
3: loop condition
4, 5: intermediate values
6: return value
7: input value

Figure 4. Factorial program in Prolog.

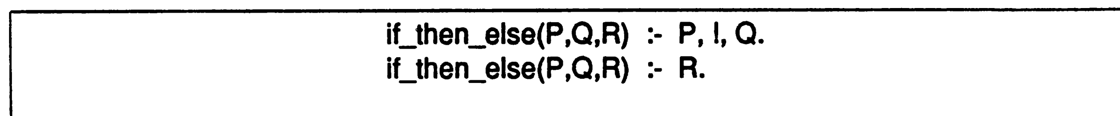


Figure 5. if_then_else/3.

1.2 Objectives

The main objective of this research is to design and to implement partially a pre-processor that is capable of transforming Prolog programs, containing simple C-like code segments, into equivalent programs that are entirely in Prolog. The pre-processor is named CIP (C in Prolog). It is assumed that CIP should be able to run in an environment in which only a Prolog compiler is available. Because of the availability and the author's familiarity with the Prolog interpreter in the Computer Laboratory at the Oklahoma State University Computer Science Department, the pre-processor is written in Quintus Prolog. The input and output of the pre-processor are in Quintus Prolog syntax.

The main benefit of the proposed pre-processor is to provide Prolog programmers with a tool that allows them to embed simple procedural constructs in Prolog programs. Thus, users could avoid some of the burden of managing data transfer between Prolog and the procedural language with which it is communicating. If the Prolog interpreter or compiler used does not have the capability to communicate with other languages, the user still has the opportunity to create procedural constructs in his or her Prolog programs. As a result, the user can rely on a Prolog compiler only.

CHAPTER II

OVERVIEW OF THE SYSTEM

2.1 Declarative Meaning of Statements in a Program

A program is a set of instructions [Struble84]. In general, an instruction is a statement indicating how the computer must control and execute its activities. The order of instructions in a program is important because it shows the order in which the computer must execute them to get the desired result. By executing the set of instructions, a computer processes the procedure in the program.

Consider a typical statement that is common in high-level languages:

$$x := 5;$$

This instruction is an assignment statement that assigns the value 5 to a variable x. Procedurally, this instruction is interpreted as "assign 5 to x". In a similar fashion, the statement

$$x := x + 3;$$

can be interpreted as "add the value 3 to the current value of variable x and then assign the result to x". These two statements can be combined to form a small program that adds two values, 5 and 3.

A program can be perceived, by its procedural meaning, as a flow of instructions. The flow of instructions is acknowledged by a

computer as a flow of statements indicating how to perform its job.

Now, assume that the instructions, which tell a computer how to perform its job, are not known. Instead, only the details of what the computer should achieve is available. In other words, propositions about what the machine should accomplish can be explicitly asserted. Then, the statement

x := 5;

may be treated as a proposition and be read declaratively as, "It is true that 5 is assigned to any variable *x* ". This proposition has an inherent truthfulness. Thus, such a proposition may have a logical value. Although there are systems with multivalued logic or fuzzy logic [Yager87], propositions used in this thesis are limited to those whose significant characteristic is their ability to denote a truth value of either true or false [Malpas87] (i.e., systems that obey the law of the excluded middle).

If a number of propositions are related to one another and the order of these propositions indicates which task should be completed before other tasks can be determined, then this sequence of propositions is actually a program. Formally, we follow the approach of interpreting a program as stated by Floyd [Floyd67]:

> "..... (an interpretation of a program) is an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then

associated proposition will be true at that time."

Looking back to the small program consisting of two assignment statements, we notice that the two consecutive instruction statements can be viewed as two consecutive declarative statements or propositions. The order of the propositons is important for it will determine what the final value of *x* will be.

The remainder of this section discusses a method of transforming the small example program into one possible Prolog program.

Prolog is a logic programming language that deals with relations rather than functions [Sethi89]. Prolog's basic notion is that it will try to prove whether a relation, given as a query, exists in the database or can be proven using rules and facts available in the database. The C statement

x = 5;

can be seen as a relation where *x* and *5* are the components of the relation, and '=' is the operator of the relation. The operator '=' can also be viewed as an assignment predicate indicating the relationship between its arguments, *x* and *5*.

This assignment relation may be transformed into Prolog notation without changing its meaning as follows

assign(X,5)

where *assign* is a predicate showing the relationship of its arguments or components, *X* and *5* . This *assign* relation is a binary relation because it defines a relation between two objects. The word *assign* is chosen instead of '=' in representing the assignment relation in Prolog because '=' is a special symbol used to define a built-in predicate that tests if its two arguments unify with one

another. The second statement in the small program

x = x + 3;

has two relational operators, '=' and '+', which indicate relations of assignment and addition. This statement can be written as

assign(X, +(X,3))

The relation *+(X,3)* is nested into the *assign* relation showing that the *addition* relation has a lower precedence than the *assign* relation. The complete form of the small program in Prolog is

```
small_program :-
    assign(X, 5), assign(X, +(X, 3)) .
```

which is also a Prolog rule. The literals before the symbol ':-' are called the head and the literals following ':-' are the body of the rule. The symbol ':-' itself is called the neck of the rule. The body is composed of two goals, *assign(X,5)* and *assign(X,+(X,3)).* The symbol ',' between the two goals indicates a conjunction of the goals. The conjunction of goals in the body of this clause reflects the conjunction of the original assertions about what the computer should do. The first assertion is the one next to the 'neck' (:-) of the clause. The original procedural program is treated as if it had a true logical value. Thus, if the two *assign* relations are true, it implies that *small_program* is true.

After the *small_program* is interpreted or compiled to the Prolog's database, a user may ask Prolog to prove that the *small_program* is true, which mean the two relations hold. In order to prove them, at least one fact or rule for *assign/2* is available in the database. Otherwise, the pre-processor, which transforms a Prolog program with embedded C code into a Prolog program, should

provide the facts and rules to guarantee that the relation is true. As far as this work is concerned, this condition is necessary because, as mentioned earlier, statements in a procedural language, like C or Pascal, are treated as if they had a true logical value. More discussion about a group of predefined rules that is known as a predefined database is provided in the last section of this chapter.

In Prolog, once a variable is instantiated or unified with a value, the value of this variable remains the same until the evaluation of the query is completed or fails. The value of the variable can be altered only when backtracking takes place. For this reason, the argument *X* in the predicate *assign* and '+' needs to be modified. If *X* in *assign(X,5)* is instantiated to a value, then *X* on the second *assign* relation and on the '+' relation will also have the same value. This condition is not what is desired because *X* on the second *assign* predicate should contain the result of the addition of *3* and *5*. To overcome this problem, the variable *X* is replaced with a predicate *x/1* whose argument indicates the variable's current value. *x/1* behaves as a global variable and is stored in the database. Every time *x/1* has a new value, its old value must be eliminated from the database. The elimination can be performed by using the built-in predicate *retract/1*. The new value can be put in the database by using the built-in predicate *asserta/1*. The modified version of *small_program* clause is

```
small_program :-
        assign(x(_1), 5),   assign(x(_2), +(x(_3), 3)).
```

2.2 The Pre-Processor's Components

In building the pre-processor, the author has been influenced by [Sterling86] and [Warren80]. There is a significant difference between the work they mentioned and the work done in this thesis. They described a simple compiler written in Prolog which translates a high level language into an assembly language. On the other hand, the pre-processor built in this thesis employs a programming language, Quintus Prolog, that is also used by the pre-processor's input and output. Since the pre-processor itself is a program and treats other programs as data, it can be called as a meta-program [Abramson89b].

The pre-processor consists of three major components (see Figure 6): Tokenizer, Embedded C Recognizer (ECR), and Clause Generator. The functions of these components are similar to the functions of components that can be found in a typical compiler [Aho86].

The top procedure of the pre-processor is *cip/2* :

```
cip(Infile, Outfile)    :-
        see(Infile),    tell(Outfile),
        tokenizer(Infile, Lexout),   % Tokenizer
        ecr(ECR_out,Lexout, []),     % Embedded C Recognizer
        clause_generator(ECR_out),      % Clause Generator
        told, seen.
```

Infile indicates the name of a file containing an input program. The output of Clause Generator will be available in an output file whose name is indicated by *Outfile* . The Tokenizer produces an output that will be passed to the Embedded C Recognizer through the the argument *Lexout*. The Embedded C Recognizer produces an output that will be given to the Clause Generator through the argument *ECR_out*. The Clause Generator produces the final output available

in the output file. The definition of the Tokenizer and the Clause Generator are given in chapter III. The discussion of the Embedded C Recognizer is available in chapter IV.

Tokenizer acts as a front-end module for the pre-processor. It accepts a stream of characters stored in a file . The function of this module is similar to what is mentioned in [Aho86]: it groups the input characters into a list of tokens that represent patterns of the input stream. ASCII codes is used for representing the Tokenizer's output.

The ECR (Embedded C Recognizer) parses the string of tokens then puts them into a form that represents the structure of the source program. If syntax errors are found, then error messages will be produced and the transformation process will not proceed at this stage. Parsing is accomplished by using a left-corner, bottom-up method [Aho72,Matsumoto83]. This algorithm has an advantage over a top-down parsing method because it can deal with left recursive rules, which appear in the Prolog grammar, without first transforming the grammar into a right recursive one. Two conditions still apply, however: the grammar must be cycle-free and must have no e-productions. Semantic analysis can be done simultaneously with syntax analysis.

The Clause Generator is the back-end module for the pre-processor. It processes the output of ECR and produces the final output of the transformation that is combined with a database of predefined clauses:

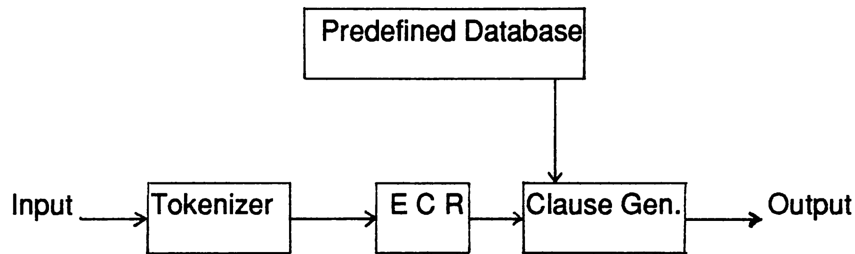


Figure 6. The Pre-Processor Block Diagram.

2.3 Input and Output

Figure 7 shows an example of a simple Prolog program containing two C assignment statements that can be recognized by the pre-processor. The intermediate output of each component is also shown together with the pre-processor's final output. Special keywords, *export* and *import*, are used to distinguish variables that are local to the embedded C code from variables in the Prolog part. Variables with the *import* designation are input variables for the embedded C code and must be instantiated to integer before entering the C code. Variables with the *export* designation contain return values and must not have been instantiated when they are used inside the embedded program. At the current stage of the work, all of these variables are assumed to be of type integer.

Another example is a Fibonacci program that calculates a value for Result as the Nth Fibonacci number. This program is written as one rule in which the body consists of six goals. The first goal of the body is the main part of the program because it is where the calculations take place. This goal shows a more complex C code fragment that can be embedded in a Prolog program (Figure 8). The output of transforming the above Fibonacci program consists

of two clauses (Figure 9).

A keyword *c_prog_n* is always used to name the entry point of the output of the transformation of the embedded procedural program. In an actual program, the last character of *c_prog_n*, *n*, is an integer showing the n-th embedded C segment appearing in a Prolog rule. The initial integer value of *n* in each rule is 1. For example, see Figure 9. C segments inside a rule are isolated segments, that means they are not related directly to other C segments available in the other rules.

The number of arguments for the predicate *c_prog_n* depends on the number of variables that are imported and exported. As an example, the predicate *c_prog_1* in the transformed Fibonacci program (Figure 9) has two arguments representing the variables N and Result.

```
addx(X) :-
          {
              export  X;
              X = 5;
              X = X +3;
          },
          write(X),  nl.
```

```
[97, 100, 100, 120,  40,  88,  41,  58, 45,  10, 123,  10,
 101, 120, 112, 111, 114, 116, 32, 88, 59, 10, 88,  61, 53,
 59, 88,  61, 88,  43,  51,  59,  10, 125,  44, 10,119, 114,
 105, 116, 101,  40,  88,  41,  44,  32, 110, 108,  46,10]
```

```
clause(head(addx,  [X]),  [c_prog_1(export(X),
                                 assign(X, 5),
                                   assign(X, +(X,3))),
                           goal(write, [X]),
                           goal(nl, [])
                         ] ).
```

```
:- [predefined_dtbase].
addx(X) :-
          c_prog_1(x(X)),
          write(X),  nl.

c_prog_1(x(X))  :-
          assign(x(_1),  5),
          assign(x(_2),  +(x(_3),  3)),  !.
```

Figure 7. The transformation process.

```
fibonacci(N,Result) :-
        {
            import  N;
            export  Result;

            fc = 1;
            if (N > 1)  {
                 Fa = 1;
                 Fb = 1;
                 for ( i = 2; i <= N; i=i+1) {
                        fc = Fa + Fb;
                        Fa = Fb;
                        Fb = fc;
                 }
            }
            Result = fc;
        },
        write(N), nl,
        write(Result), nl.
```

Figure 8. A Prolog program with embedded C code.

```
fibonacci(N,Result) :-
        c_prog_1(n(N), result(Result)),
        write(N), nl,
        write(Result), nl.

c_prog_1(n(N),  result(Result)) :-
        integer(N),
        asserta(n(N)),
        assign(fc(_1), const(1)),
        if_then(  greater_than( n(_2), const(1)  ),
            [  assign( fa(_3),   const(1)),
            assign(fb(_4),  const(1)),
            for_loop(
                 assign( i(_5),  const(2)),
                 less_or_equal(  i(_6),  n(_7)  ),
                 assign(  i(_8),  +( i(_9),  const(1)  )  ),
                 [   assign(fc(_10), +(fa(_11),  fb(_12) ) ),
                     assign( fa(_13),  fb(_14)),
                     assign( fb(_15),  fc(_16))
                 ]
                 )
            ]
        ),
        assign( result(Result),  fc(_17) ), !.
```

Figure 9. The result of transforming the program in Figure 8.

2.4 Predefined Database

The predefined database contains the clauses that are needed by the output of the pre-processor as a consulted database. These clauses can be divided into six categories (shown in Appendix A): assignment clauses, conditional clauses, relational-operator clauses, Boolean-operator clauses, and clauses that deal with expressions and values of identifiers and constants. Each of these categories of clauses corresponds to the C operators (=, ==, !=, <, >, =<, >=, &&, ||, +, -, *, /) and to the C conditional statements ('if', 'while', 'for').

Assignment clauses consist of three procedures: *assign/2*, *expr_value/2*, and *value/2*. A procedure is a set of rules with the same predicate in the head of the rules [Sterling86]. The first argument of the predicate *assign* is always matched with a term of one arity, e.g., y(4). The functor of this term, y, represents a variable name found in the embedded procedural construct of the pre-processor's input program. The argument of the term, 4, represents the current value of the variable. Whenever the current value of a variable is replaced by a new value, a Prolog fact that represents the variable and its value (i.e., term) is retracted from the database. A new Prolog fact is, then, asserted representing the variable with its new value. The second argument of the predicate *assign* represents a constant or an expression. An expression is represented by *Expr* whose value is determined by the predicate *expr_value(Val,Expr)* . This predicate calculates *Val* as the value of the expression *Expr* . An expression can be in the form of a constant, identifier, addition, multiplication, subtraction, division, or a

combination of them. The combination is necessary because in Prolog, embedded relations are not evaluated, as they would be in C.

The rule :

expr_value(Val,+(Expr1,Expr2)) :-

value(A,Expr1), value(B,Expr2), Val is A + B .

is read as:

Val is the value of *Expr1* plus *Expr2* if *A* is the value of *Expr1* and *B* is the value of *Expr2* and *Val* is the sum of *A* and *B* .

The next group of clauses to be discussed is the conditional clauses. The purpose of these clauses is to duplicate the behaviour of if-then, if-then-else, while-loop, and for-loop statements of procedural languages. If there is a goal in a target program that matches the head of one of these clauses, it is guaranteed that the goal will succeed.

The two predicates, *if_then/2* and *if_then_else/3*, are used to simulate the 'if' statement. Both of these predicates have a *Cond* argument and a *True_body* argument. *Cond* is used to test whether the condition of the 'if' statement is logically true or false. If the condition is logically true, the goal *if_true_body/1*, in which its argument *[Statement|Rest_st]* has been unified with *True_body* will be the next to be attempted. On the other hand, if *Cond* is false, the second clause of *if_then/2* or *if_then_else/3* is processed. In the case of *if_then_else/3*, its third argument, *False_body* will be unified with *[Statement|Rest_st]*, the argument of *if_false_body/1*. The predicate *for_loop/4* replicates the function of a 'for-loop' statement. The argument *Init* initializes the counter that is incremented or decremented by *Count*. After initialization, the predicate *forloop1/3* takes care of checking whether the

condition is met, call *forbody/1* to execute statements inside the loop, and increment or decrement the counter.

The predicate *while_loop/2* behaves as if it were a while-loop statement. *Cond* checks whether the condition is valid. After the goal *wh_body/1* succeeds, *while_loop/2* calls itself in order to repeat the loop until *Cond* fails. The second clause of *while_loop/2* is necessary to guarantee that the predicate is always true even if *Cond* fails. *while_loop/2* must be true because the code corresponding to C code must succeed.

for_loop/4 and *while_loop/2* are recursive procedures. Recursive procedures in Prolog may consumes a lot of time and space for maintaining stack. Fortunately, Quintus Prolog used in this thesis provides tail recursion optimization. Tail recursion optimization is a way of economizing on the amount of stack space needed to evaluate a query to a procedure by utilizing a constant number of stack units [Malpas87,Quintus87].

Relational-operator clauses are used to check whether the relational operators (==, !=, <, >, =<, or >=) hold. Each clause in this group represents one of those relational operators. After both expressions in a relation are evaluated, the corresponding relational-operator clause compares them according to its relational operator. If the comparison does not hold, the clause simply fails.

Similar to relational-operator clauses, Boolean-operator clauses are used to test whether a Boolean expression is true. There are three rules to examine Boolean expressions: AND, OR, and NOT. It should be noted that the arguments *Relexpr1* and *Relexpr2* can be Boolean expressions.

CHAPTER III

TOKENIZER AND CLAUSE GENERATOR

3.1 Tokenizer

Tokenizer is the front-end module of CIP, the pre-processor designed and partially implemented in this thesis. The main purpose of the module is to provide a sequence of tokens that can be parsed by the subsequent module, ECR. These tokens represent an input program.

In general, a token is a string that can be used to represent the string itself or a set of strings which are described by a rule called a pattern. Aho et al.[Aho86] discuss various techniques of specifying and recognizing tokens. They mention that regular expressions are an important notation for specifying patterns. Regular expressions are used for defining sets of strings that match with certain patterns. In order to recognize tokens, a deterministic transition diagram may be applied. The diagram is useful in keeping of information about characters that are read as the pointer scans the input. The transition diagram, then, can be converted into a program to find the tokens specified by the diagram.

Since the Tokenizer in this thesis was expected to produce relatively simple tokens, a discussion of regular expressions and transition diagrams is not included here. An in-depth discussion can be found in [Aho86].

Two things must be considered in order to build the Tokenizer:

1. The tokens produced must represent the corresponding input strings correctly.
2. The Tokenizer should be able to do its job efficiently.

The first consideration is necessary to provide the Embedded C Recognizer (ECR) with a correct internal representation of an input program. As an example, suppose an input program contains the following sequence of characters that represents a clause:

n(X) :- X is 5.

The Tokenizer must be able to acknowledge that the last character, '.', is a full-stop character instead of a decimal-point. Furthermore, spaces between characters should not be ignored; otherwise the produced tokens may represent the input strings incorrectly, as the following example shows:

n (X) :- X is 5.

The rule is syntactically incorrect because there is a space between the predicate 'n' and '('. If spaces between the group of strings are ignored, the generated tokens will be 'n', '(X)', ':-', 'X', 'is', '5', and '.' which may cause the ECR to interpret these tokens as representing a syntactically correct clause.

The second consideration concerns the reduction of unnecessary processing time used by the Tokenizer. We need the module to do its job without creating output that will not be used by the ECR. Thus, execution time may be saved and can be allocated to other modules that require considerably more processing time.

needed to produce outputs for debugging purposes. These mechanisms have been deleted from the final form of the Tokenizer.

Based on the two considerations described above, the author has decided that every character in an input program will be represented by a single token. A string that is composed of 5 characters, for example, will be represented by 5 tokens in the corresponding order. Since each of the input characters is represented by a distinct token, the Tokenizer is not necessary to spend time for selecting and grouping them into specific tokens. The produced tokens are collected into a list.

```
%          +       -
tokenizer(Filein, Clistout) :-
      see(Filein),                    % open an input file
      Clist = [],                     % unify Clist and []
      readfile(Clist, Clist1),        % read the input file
      reverse(Clist1, Clistout),
      seen, !.

%          +       -
readfile(Clist, Clistout) :-
      get0(Char),
      ( Char == -1,      % eof = -1 [Quintus87]
        ( Clistout = Clist ; readfile([Char|Clist], Clistout) )
      ).

% List2 is List1 in reversed order
%          +       -
reverse(List1, List2) :-
      rev(List1, [], List2).
rev([], L, L).
rev([X|L1], L2, L3) :-
      rev(L1, [X|L2], L3).
```

Figure 10. Tokenizer.

3.2 Clause Generator

Clause Generator is the back-end module of the preprocessor and is responsible for producing the final output. It converts a list of ASCII character codes given by the ECR into a set of printable characters. Unprinttable ASCII codes, such as 'newline' character, are treated as such. Before the conversion begins, the Clause Generator will assert a command for consulting the predefined rules. Figure 11 shows the program of the Clause Generator.

```
%                +
clause_generator(L) :-
        tell(cg_out),
        write(':- [predefined_rules]. '), nl,
        out(L),
        told.

% +
out([X|Xs]) :-
        name(M,[X]),
        (X == 10, nl;  write(M)),
        out(Xs).
out(_) :- true.
```

Figure 11. Clause Generator.

CHAPTER IV

EMBEDDED C RECOGNIZER

Embedded C Recognizer (ECR) is a module in CIP that has the responsibility of recognizing the existence of C constructs in a Prolog program. In order to accomplish its task, ECR must be able to distinguish the structure of the embedded C statements from the structure of the Prolog program in which it is embedded. In other words, it has to be equipped with a parser mechanism that is capable of recognizing the different structures of the two programming languages. By providing the syntax rules of Prolog and a subset of C for ECR, the first step of building the module is achieved. The next step is to find a parsing method suitable for the implementation.

4.1 Context Free Grammars and Left Recursion in Prolog

The syntax of Prolog used in this thesis is the one used in Quintus Prolog [Quintus87]. The choice was made because the related information on Quintus needed to support the work was more accessible than that on other Prolog versions available. Although this information is still very general, it did save some amount of searching time and was very useful in guiding the implementation of ECR.

A Prolog grammar can be described using context-free format

in which each production has the form:

```
A  -->  ß
```

The symbol A represents a non-terminal. The symbol ß represents a string of non-terminals and/or terminals, or an empty string. In a context-free grammar, only one non-terminal appears at the left-hand side of the production. Unlike non-terminals in grammars for many common programming languages, non-terminals in some Prolog grammar rules have arguments. One example of such a production is

```
term(N)    -->  op(N,fx)
```

The non-terminals of the production are 'term' and 'op' while N and fx are arguments. 'op' or 'term' represents any Prolog operator. This production indicates that a term with precedence N can be of the form op with precedence N and fx as the specifier for a prefix operator. The precedence N, usually an integer number between 1 and 1200, is used to disambiguate expressions where the syntax of the terms is not made explicit through the use of brackets. The specifier fx is one of the available specifiers in Prolog used to disambiguate expressions in which there are two Prolog operators in the expression that have the same precedence [Clocksin87].

Prolog grammar inherently has left recursion in its productions as shown by this example:

```
goals    --> goals  ,  goals
           | goals  ;  goals
           | goal
```

where 'goals' and 'goal' are non-terminals, ',' and ';' are terminals. This grammar rule shows that the non-terminal 'goal' on the left-hand side can be substituted by one of the three forms

available on the right-hand side of the rule. The first form is "goals , goals". The other two forms follows the symbol '|' which stands for the boolean 'or'. Left recursion prohibits the use of top-down parsing because it leads the parser into an infinite loop. The discussion of the parsing technique used in the implementaion of CIP is given in Section 4.5.

4.2. A Subset of C Syntax

The subset of C grammar rules (shown in Appendix C) is obtained by interpreting the syntax chart of ANSI C available in [Darnell91]. Only the basic C syntax rules were adopted for the purpose of this thesis. This choice limits the structures of C that can be embedded in Prolog programs. The focus of this work is on the recognition of C constructs, a limited number of C syntax rules are considered adequate for prototyping. Further expansion is possible and is proposed as one of the directions of future work.

4.3 Top-Down Parser and Left Recursion

Many commercial Prolog versions, including Quintus Prolog used in this thesis, have a built-in top-down parser. The parser is based on the Definite Clause Grammar (DCG) formalism [Pereira80]. A DCG is an extension of executable context-free grammars used in Prolog. Each non-terminal in a production is a predicate in the corresponding Prolog clause. By using DCG's, users not only have an executable context-free grammar, but they also have the possibility of inserting any Prolog calls within a rule. These facilities are suitable for integrating parser and semantic-processing procedures. Because of the natural top-down and left-to-right processing of

Prolog however, DCG's have the disadvantage that they cannot deal with left-recursive rules [Matsumoto83, Stabler83]. Left recursive rules cause the parser to go into an infinite loop.

A top-down parser may still be used if left-recursive rules in a grammar are eliminated. Aho and Ullman describe an algorithm for eliminating left-recursion in a proper (cycle free, e free, and useless-symbols free) Context-Free Grammar [Aho72]. However, left-recursion elimination is not a practical approach for a Context-Free Grammar having a large number of productions. Users can expect a large number of new productions created by the elimination procedure (exponentially larger than the initial set).

Fortunately, left recursion can be dealt with by a bottom-up parser despite the fact that such a parser is generally more difficult to build and less efficient than a top-down parser. It is not the purpose of this thesis to examine many bottom-up parsing techniques available to date. Prolog usage of parsing and compiling techniques can be found in [Cohen87].

4.4 Left-Corner Parsing Method for ECR

The parsing technique used in ECR is called left-corner parsing. A previous practical implementation of the technique is shown in [Matsumoto83,Matsumoto86]. The technique combines bottom-up recognition and top-down recognition. A formal definition [Aho72] follows:

> The *left corner* of a non-*e*-production is the leftmost symbol (terminal or non-terminal) on the right side. A *left-corner parse* of a sentence is the sequence of productions used at the interior nodes of a parse tree in which all nodes have

been ordered as follows. If a node *n* has *p* direct descendants *n1*, *n2*, ..., *np*, then all nodes in the subtree with root *n1* precede *n*. Node *n* precedes all its other descendants. The descendants of *n2* precede those of *n3*, which precede those of *n4*, and so forth.

As an illustration, consider the following context-free grammar G = (N,Σ,P,S) with the set of productions, P:

1. S --> AS
2. S --> BB
3. A --> bAA
4. A --> a
5. B --> b
6. B --> e

the set of non-terminals, N = {S,A,B}, and the set of terminals, Σ = {a, b}. The start symbol is S. The symbol 'e' indicates an empty string. A nondeterministic left-corner parser for G can be defined using a PDT (Pushdown Transducer) M as follows:

$$M = (\{q\}, \Sigma, N \times N \cup N \cup \Sigma, \Delta, \partial, q, S, \emptyset), \text{ where}$$

{q} is the set of possible states,

Σ is a finite set of input alphabet,

N is a finite set of non-terminals,

N x N U N U Σ is a finite alphabet of pushdown list symbols,

Δ = {1,2,3,4,5,6} is the set of output alphabet (showing the production numbers),

q is the initial state,

S, the start symbol, is the symbol that appears initially on the pushdown list,

ø indicates an empty set of final states,

∂ , the transition function, is a mapping from $\{q\} \times (\Sigma \cup \{e\}) \times (N \times N \cup N \cup \Sigma)$ to finite subsets of $\{q\} \times (N \times N \cup N \cup \Sigma)^* \times \Delta^*$ and has the form:

$$\partial(q_i, c, C) = \{(q_j, D, O_1), (q_k, H, O_2), \ldots\} \qquad i, j, k \geq 0$$

The transition function shows that there are two or more possible transitions when the automaton is in state q_i scanning the current input symbol c with C on the top of the pushdown stack. (qj, D, O_1) represents the new state q_j, the new symbol D on top of the stack, and the emitted-output symbol O_1. ∂ for the contxt-free grammar G is defined as follows for all X in N:

1 a. $\partial(q, e, [X,A]) = \{(q, S[X,S], 1)\}$.
  b. $\partial(q, e, [X,B]) = \{(q, B[X,S], 2)\}$.
  c. $\partial(q, e, X) = \{(q, bAA[X,A], 3),$
  d. $(q, a[X,A], 4),$
  e. $(q, b[X,B], 5),$
  f. $(q, [X,B], 6)\}$.
2. $\partial(q, e, [A,A]) = \{(q,e,e)\}$.
3. $\partial(q, c, c) = \{(q,e,e)\}$ for all c in Σ

Special symbols of the form ß[P,Q], where P and Q are non-terminals, indicates that P is the current goal to be recognized and Q is the non-terminal which has just been recognized bottom-up. Every ß, an element of $(N \cup \Sigma)^*$, is a symbol representing goals to be recognized top-down. More detailed description of defining ∂ for a left corner parser appears in [Aho72].

The parser can be used now to parse a string, e.g., *bbaaab* . The parse tree for this string is shown in Figure 12 Let the initial configuration for the left-corner parser be: (bbaaab,S,e), where bbaaab is the portion of input to be parsed, S is the initial contents of the top of the stack, and e or 'empty' indicates the initial output. Then, the parser will go into one possible sequence of configurations, as shown in Figure 13, that successfully parses the input string.

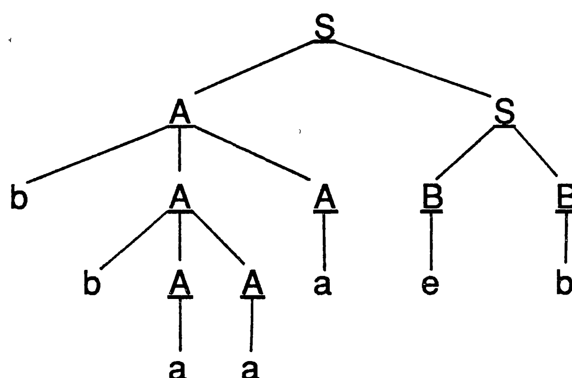


Figure 12. The parse tree for the string *bbaaab* [Aho72].

∂ applied (#)	next configuration
1c	(bbaaab, bAA[S,A], 3)
3	(baaab, AA[S,A], 3)
1c	(baaab, bAA[A,A]A[S,A], 33)
3	(aaab, AA[A,A]A[S,A], 33)
1d	(aaab, a[A,A]A[A,A]A[S,A], 334)
3	(aab, [A,A]A[A,A]A[S,A], 334)
2	(aab, A[A,A]A[S,A], 334)
1d	(aab, a[A,A][A,A]A[S,A], 3344)
3	(ab, [A,A][A,A]A[S,A], 3344)
2	(ab, [A,A]A[S,A], 3344)
2	(ab, A[S,A], 3344)
1d	(ab, a[A,A][S,A], 33444)
3	(b, [A,A][S,A], 33444)
2	(b, [S,A], 33444)
1a	(b, S[S,S], 334441)
1f	(b, [S,B][S,S], 3344416)
1b	(b, B[S,S][S,S], 33444162)
1e	(b, b[B,B][S,S][S,S], 334441625)
3	(e, [B,B][S,S][S,S], 334441625)
2	(e, [S,S][S,S], 334441625)
2	(e, [S,S], 334441625)
2	(e, e, 334441625)

Figure 13. The sequence of configurations for parsing the string *bbaaab*.

4.5 Left-Corner Parsing Implementation in Prolog

This section discusses how context-free production rules can be transformed into Prolog clauses in order to determine the behavior of a left-corner parser [Abramson89, Matsumoto83]. The parser can deal with any context-free grammar having no cycle and no e-productions.

Suppose context-free productions are represented in the following forms:

1. B —> c
2. B —> X1, X2, ..., Xn ($n \geq 1$).

where B and c represent a non-terminal symbol and a terminal symbol respectively, and Xi,$1 \leq i \leq n$, are either non-terminal or terminal symbols. A production rule with only one terminal symbol on the right-hand side, such as B --> c, is transformed into a unit clause:

```
dict(B, [c], [c|X], X).
```

The unit clause is called a dictionary rule. Its first argument indicates the left-hand side of the production rule. The second argument indicates the subtree generated by the rule. The third and fourth arguments represent the difference-list [Pereira80] of the input string in which the first item must be a terminal. A difference-list is a way of representing a sequence of elements. For example, a sequence of A,B,C can be represented by the difference of pair of lists L1= [A,B,C,D,E] and L2 = [D,E], or, L1= [A,B,C] and L2 = []. [] means an empty list.

The second form of the context-free production is translated into two forms of Prolog clauses:

```
x1(x1,T,T,X,X).                          %1
x1(Goal,T1,Tout,X0,Xn)   :-              %2
    link(b,Goal),
    gl(x2,T2,X0,X1),
    gl(x3,T3,X1,X2),
    ...
    ...
    gl(xn,Tn,Xn2,Xn1),
    b(Goal,Tn1,Tout,Xn1,Xn).
```

Xn2 and Xn1 represent X_{n-2} and X_{n-1}, respectively. The description of the arguments of the predicate x1/5 is as follows:

Goal: input, the 'goal' non-terminal to be accomplished during parsing

Tout: output, the part of the derivation tree created after successfully using the rule

T1: output, a sub-derivation tree representing the non-terminal x1

X0 and Xn: input, represent the difference list of the input string.

If a 'goal' non-terminal x1 is given and the non-terminal called is also x1, the call terminates successfully by rule %1; otherwise, if the 'goal' non-terminal is not x1, rule %2 is used.

The predicate link(A,Goal) determines whether there is a reflexive and transitive relation between the two non-terminals, A and Goal. A non-terminal N1 has a 'link' relation with a non-terminal N2 if there is a grammar rule whose form is "N2 --> N1, ...". If the relation exists, the predicate gl/4 is called in order to recognize X2, ..., Xn. The definition of the predicate gl/4

is

```
gl(G,A,X,Z) :-
        dict(C,A1,X,Y),  P =..  [C,G,A1,A,Y,Z],  call(P).
```

An example of an implementation of the parsing method is in parsing a natural language, e.g., English [Matsumoto86]. Suppose the parser is given the simple English grammar:

```
sentence     --> nounphrase ,  verbphrase.
nounphrase   --> [calvin].
verbphrase   --> [yawns].
```

The Prolog clauses which implement the left-corner parsing method described above are:

```
% dictionary
dict(nounphrase, [calvin], [calvin|X], X).
dict(verbphrase, [yawns], [yawns|X], X).

% rule
nounphrase(G,[NP,N],A,X,Z)   :-
        link(sentence,G),
        gl(verbphrase,[VP,N],X,Y),
        sentence(G,[s(NP,VP)],A,Y,Z).

%  terminate clauses
nounphrase(nounphrase,T,T,X,X).
verbphrase(verbphrase,T,T,X,X).
sentence(sentence,T,T,X,X).

%  'goal' clause
gl(G,A,X,Z)   :-
        dict(C,A1,X,Y),
        P  =..  [C,G,A1,A,Y,Z], call(P).
```

Using the above clauses, Prolog is able to give an answer 'yes' to a query such as "?- gl(sentence,[calvin,yawns],[]).".

4.6 Static Semantic Analysis

The tasks of the Embedded C Recognizer (ECR) are syntax analysis, semantic analysis, and C-to-Prolog translation. Sections 4.4. and 4.5 describe the left-corner method employed by ECR to perform syntax analysis. This section briefly discusses the design of the semantic analysis. A discussion of C-to-Prolog translation is given in the next section.

Similar to the tasks of semantic routines available in a typical compiler, semantic analysis performed by ECR assures the validity of the static semantic of each embedded C-like statement. Examples of static semantic checking are type checking, flow-of-control checking, uniqueness checking, and name-related checking [Aho86]. ECR also employs Attribute Grammar, a popular method for formalizing the specification of static semantics. An attribute grammar augments ordinary context-free grammars with values that represent the semantic property of a symbol. The following context-free grammar rule, written in DCG notation, shows the attributes attached to its corresponding non-terminals:

```
expr(E1val) --> expr(E2val), '+', term(Tval), {E1val is E2val + Tval}.
```

The literal between curly brackets is called an attribute rule. It evaluates the attribute value E1val. In an actual implementation, a grammar production may have more than one attribute rule.

There are two types of attributes: synthesized and inherited. In the grammar rule above, E1val is a synthesized attribute because it appears on the left-hand side and it is computed from the values of attributes on the right-hand side of the production. E2val and

Tval are inherited attributes since they appear on the right-hand side of the production. Their values can be computed from the values of attributes available on either side of the production. Synthesized attributes are used to pass information up a syntax tree, while inherited attributes are used to pass information down a syntax tree [Fischer91]. Terminal symbols may have only synthesized attributes. Non-terminal symbols may have both of them. All inherited attributes of the start symbol are recognized as initial values.

What follows is the approach used in the design of the semantic routines for ECR:

- Associating a grammar symbol with a semantic record. Each different grammar symbol will have a distinct record containing information appropriate for that symbol. Semantic routines produce data on which they may operate. The data is named semantic information. It is possible for a symbol to have no semantic record.
- Defining semantic records. This is done by examining each symbol in the context-free grammar and deciding what, if any, semantic information needs to be stored for that symbol. Deciding the information on semantic records means deciding what parameters a semantic routine will have.
- Adding semantic rules to the grammar in order to specify when semantic processing should take place.

4.7. C-to-Prolog Translation

The final task of the Embedded C Recognizer, after successfully performing syntax and semantic analysis, is translating the embedded C code into Prolog-like structure. There

are two times at which the translation can be performed. First, translation may be accomplished every time a valid embedded C statement is recognized. The advantage of this choice is that ECR does not have to keep the information about the statement after the translation is completed. The disadvantage is that ECR may redo the translation because it employs a nondeterministic parsing method. Second, translation can be performed once, that is, after all embedded C statements are recognized. This choice gives the advantage of avoiding possible repeated translation, but ECR must keep all information until the translation is performed.

A method of translation using Prolog can be found in [Warren80]. Although the method is used to translate a high-level language into an assembly language, it can be modified in order to be incorporated in ECR. The modification proved to be more substantial than expected, therefore it is relegated as potential future work.

CHAPTER V

SUMMARY AND SUGGESTED FUTURE RESEARCH

A pre-processor that is capable of transforming simple Prolog programs, in which basic C statements can be integrated in the body of clauses, has been presented. The pre-processor, named CIP, transforms such a program into a program that is entirely in Prolog. Each embedded C statement is treated as a true assertion. In order to guarantee that this assertion is always logically true, a set of supporting rules are defined.

The design of CIP includes the Tokenizer, the Embedded C Recognizer (ECR), and the Clause Generator. The Tokenizer serves as the front-end module of CIP and produces tokens that represent an input Prolog program. The Embedded C Recognizer processes the tokens to check whether there are embedded C structures that are syntactically and semantically valid in the input program. If valid embedded C structures are found, ECR translates the input program into an intermediate output. Using the intermediate output, the Clause Generator generates the final output. The final output is then ready to be compiled by the available Prolog compiler.

The implementation of CIP assumes that a Prolog compiler is the only compiler available. Thus, the Tokenizer and the Clause Generator are implemented in Prolog. Partial implementation of ECR is also carried out in Prolog. The left-corner parsing method is used to recognize embedded C structures in Prolog programs.

Multiple embedded C segments in the body of a Prolog clause can be recognized by ECR. The implementation of semantic analysis and intermediate output translation are perceived to be beyond the scope of this thesis and are considered as important future research.

Three benefits of the pre-processor may be mentioned:

1. It provides Prolog programmers the opportunity to integrate procedural constructs into logic programs.
2. In the case of simple procedural programs, the burden of managing the data transfer and declaring the modules required for linking multiple languages can be avoided.
3. Because of the fact that CIP is composed of modules which are built based on their functionalities, this pre-processor can be used as a practical and educational environment for implementing many different theoritical algorithms of syntax/semantic analysis, code generations, and pattern recognition.

The limitation of CIP is that it can only recognize simple C statements such as while-loop, for-loop and if-then-else statements. All variables in the embedded C program are assumed to be of type integer.

Further improvements to CIP are possible. It may be improved by adding features which allow complex C statements, such as recursive calls and the use of array, to be embedded in a Prolog program.

REFERENCES

[Abramson89a]
Abramson, H. and Dahl, V., Logic Grammars, Springer-Verlag, New York, 1989.

[Abramson89b]
Abramson, H. and Rogers, M.H. (Eds.), Meta Programming in Logic Programming, MIT Press, Cambridge, MA, 1989.

[Aho72]
Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translation, and Compiling, Volume I: Parsing, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Aho86]
Aho, A.V., Sethi, R., and Ullman, J.D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

[Arity88]
The Arity/Prolog Language Reference Manual, Arity Corp., Concord, MA, 1988.

[Bratko86]
Bratko, I., Prolog Programming for Artificial Intelligence, Addison-Wesley, Reading, MA, 1986.

[Clocksin81]
Clocksin, W. F. and Mellish, C. S. , Programming in Prolog, Springer-Verlag, Berlin, Heidelberg, New York, 1981.

[Cohen87]
Cohen, J. and Hickey, T.J., "Parsing and Compiling Using Prolog," ACM Transactions on Programming Languages and Systems, vol. 9, no. 2, 1987, pp.125-163.

[Colmerauer85]
Colmerauer, A., "Prolog in 10 Figures," CACM, vol. 28, no.12, December 1985, pp. 1296-1310.

[Colmerauer90]
Colmerauer, A., "An Introduction to Prolog III," CACM, vol. 33, no. 7, July 1990, pp. 69-90.

[Darnell91]
Darnell, P.A. and Margolis, P.E., C: A Software Engineering Approach, Springer-Verlag, New York, 1991.

[Fischer91]
Fischer, C.N. and LeBlanc, R.J., Crafting a Compiler with C, Benjamin/Cummings, Redwood City, CA, 1991.

[Floyd67]
Floyd, R. W., "Assigning Meanings to Programs," Proceedings of the Symposium in Applied Mathematics (J.T. Schwartz, Ed.), vol.19, American Mathematical Society, Providence, R.I., 1967, pp.19-32.

[Grishman86]
Grishman, R., Computational Linguistics: An Introduction, Cambridge University Press, Cambridge, UK, 1986.

[Kernighan78]
Kernighan, B.W. and Ritchie, D.M., "The C Programming Language," Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[Kowalski79]
Kowalski, R., "Algorithm = Logic + Control," CACM, vol. 22, no. 7, July 1979.

[Lloyd84]
Lloyd, J.W., Foundations of Logic Programming, Springer-Verlag, New York, NY, 1984.

[Malpas87]
Malpas, J., PROLOG: A Relational Language and Its Applications, Prentice Hall, Englewood Cliffs, NJ, 1987.

[Maruyama84]
Maruyama, H. and Yonezawa, A., "A Prolog-Based Natural Language Front-End System," New Generation Computing 2, 1984, pp. 91-99.

[Matsumoto83]
Matsumoto, Y. et al., "BUP: A Bottom-Up Parser Embedded in Prolog," Journal of New Generation Computing, OHMSHA, Ltd., vol. 1, no.2, 1983, pp. 145-158.

[Matsumoto86]
Matsumoto, Y., Tanaka, H., and Kiyono, M., "BUP: A Bottom-Up Parsing System for Natural Languages," in Logic Programming and Its Applications (Caneghem, Michel van and Warren, D.H.D, Eds.), ABLEX Publishing Corp., Norwood, NJ, 1986.

[Pereira80]
Pereira, F.C.N. and Warren, D.H.D, "Definite Clause Grammars for Language Analysis: A Survey of the Formalism and a Comparison with Augmented Transition Networks," Artificial Intelligence 13, 1980, pp. 231-278.

[Quintus87]
Quintus Prolog Reference Manual Version 10, Quintus Computer Systems, Inc., Mountain View, California, 1987.

[Roach90]
Roach, D. and Berghel, H., "A Mixed-Language Expert System," PC-AI, Sept./Oct.1990, pp. 46-48.

[Santoso91]
Santoso, L.B., Mayfield, B.E., and Samadzadeh, M.H., "Embedding C Constructs in Prolog," Proceedings of the First Golden West Conference on Intelligent Systems, June 3-5, 1991, Reno, Nevada.

[Sethi89]
Sethi, R., Programming Languages: Concepts and Constructs, Addison-Wesley, Reading, MA, 1989.

[Stabler83]
Stabler, E.P., Jr., "Deterministic and Bottom-up Parsing in Prolog," Proceedings of the National Conference on Artificial Intelligence, Washington, DC, AAAI-83, August 22-26, 1983, pp. 383-386.

[Sterling86]
Sterling, L. and Shapiro, E., The Art of Prolog: Advanced Programming Techniques, MIT Press, Cambridge, MA, 1986.

[Struble84]
Struble, G., Assembler Language Programming, Addison-Wesley, Reading, MA, 1984.

[Thayse88]
Thayse, A., From Standard Logic to Logic Programming, John Wiley & Sons, New York, 1988.

[Uehara83]
Uehara, T. and Kawato, N., "Logic Circuit Synthesis Using Prolog," New Generation Computing 1, Ohmsa Ltd, Japan, 1983, pp. 187-193.

[Waldinger74]
Waldinger, R. J. and Levitt, K. N., "Reasoning about Programs," Artificial Intelligence 5(3), 1974, pp. 235-316.

[Walker90]
Walker, A., McCord, M., Sowa, J. F., and Wilson, W.G., Knowledge Systems and Prolog, Addison-Wesley, Reading, MA, 1990.

[Warren80]
Warren, D.H.D., "Logic Programming and Compiler Writing," Software Practice and Experience, vol. 10, no.2, 1980, pp. 97-125.

[Warren82]
Warren, D.H.D. and Pereira, F.C.N., "An Efficient Easily Adaptable System for Interpreting Natural Language Queries," American Journal of Computational Linguistics 8, 1982, pp. 110-122.

[Wojciechowski83]
Wojciechowski, W.S. and Wojcik, A.S., "Automated Design of Multiple-Valued Logic Circuits by Automated Theorem Proving Techniques," IEEE Transactions on Computers, vol. C-32, no. 9, Sept. 1983, pp. 785-798.

[Yager87]
Yager, R.R., Ovchinnikov, S., Tong, R.M., and Nguyen, H.T., Fuzzy Sets and Applications: Selected Papers by L.A. Zadeh, John Wiley & Sons, Inc., Canada, 1987.

APPENDIXES

APPENDIX A

GLOSSARY

CIP: The name of the pre-processor which is designed and implemented partially in this thesis. CIP stands for C in Prolog.

Clause Generator: Sometimes called Code Generator, a component of the pre-processor that obtains input from the Embedded C Recognizer and produces a set of clauses as a target program.

Embedded C Recognizer (ECR): Sometimes called Syntax Analyzer, a component of the pre-processor that obtains a string of tokens from the Lexical Analyzer and verifies that the string represents a Prolog program with embedded C statements.

Tokenizer: Sometimes called Lexical Analyzer or Scanner, a component of the pre-processor that converts a stream of input characters into a stream of tokens.

APPENDIX B

PREDEFINED DATABASE

PREDEFINED DATABASE

```
%%--------------------
%% Assignment clauses.
%%--------------------

% assign/2
assign(X,Expr) :-
                X,
                X =.. [Pred,Arg], integer(Arg),
                expr_value(Val,Expr),
                Z =.. [Pred,Val], retract(X), asserta(Z).
assign(X,Expr) :-
                X =.. [Pred,Arg], var(Arg),
                expr_value(Val,Expr),
                Z =.. [Pred,Val], asserta(Z).

% expr_value/2.
expr_value(Val,+(Expr1,Expr2)) :-
                value(A,Expr1), value(B,Expr2), Val is A + B.
expr_value(Val,-(Expr1,Expr2)) :-
                value(A,Expr1), value(B,Expr2), Val is A - B.
expr_value(Val,-(Expr1)) :-
                expr_value(A,Expr1), Val is - A.
expr_value(Val,/(Expr1,Expr2)) :-
                value(A,Expr1), value(B,Expr2), Val is A / B.
expr_value(Val,*(Expr1,Expr2)) :-
                value(A,Expr1), value(B,Expr2), Val is A * B.
expr_value(Val,Expr1) :-
                value(Val,Expr1).

% Val is the value of a constant, an identifier , or an expression.
value(Val,const(X)) :-
                integer(X), Val is X.
value(Val,Identifier) :-
                Identifier =.. [Pred,Arg], var(Arg),
                Identifier, Val = Arg.
value(Val,Identifier) :-
                Identifier =.. [Pred,Arg], integer(Arg),
                Id =.. [Pred,Arg_1], Id, Val = Arg_1.
value(Val,Expr) :-
                expr_value(Val,Expr).
```

```
%%------------------
%% Conditional clauses.
%%------------------

% if_then/2
if_then(Cond,True_body) :-
            Cond, !,
            if_true_body(True_body).
if_then(Cond,True_body).

% if_then_else/3
if_then_else(Cond,True_body,False_body) :-
            Cond, !,
            if_true_body(True_body).
if_then_else(Cond,True_body,False_body) :-
            if_false_body(False_body).

% if_true_body/1
if_true_body([Statement|Rest_st]) :-
            Statement, if_true_body(Rest_st).
if_true_body([]).

% if_false_body/1
if_false_body([Statement|Rest_st]) :-
            Statement, if_false_body(Rest_st).
if_false_body([]).

% while_loop/2
while_loop(Cond,[Statement|Rest_st]) :-
            Cond,
            wh_body([Statement|Rest_st]),
            while_loop(Cond,[Statement|Rest_st]).
while_loop(Cond,[Statement|Rest_st]).

% while_body/1
wh_body([Statement|Rest_st]) :-
            Statement, wh_body(Rest_st).
wh_body([]).

% for_loop/4
for_loop(Init, Cond, Count, Body) :-
            Init,
            forloop1(Cond,Count,Body).

% forloop1/3
forloop1(Cond,Count,Body) :-
            Cond,
            for_body(Body), Count,
            forloop1(Cond,Count,Body).
forloop1(Cond,Count,Body).
```

```
% for_body/1
for_body([Statement|Rest_st]) :-
                Statement, for_body(Rest_st).
for_body([]).


%%---------------------------------
%% Relational-operator clauses.
%%---------------------------------
% Expr1 == Expr2
equal(Expr1,Expr2) :-
                expr_value(Val1,Expr1),
                expr_value(Val2,Expr2),  !,
                Val1 =:= Val2.

% Expr1 != Expr2
not_equal(Expr1,Expr2) :-
                expr_value(Val1,Expr1),
                expr_value(Val2,Expr2),  !,
                Val1 \== Val2.

% Expr1 < Expr2
less_than(Expr1,Expr2) :-
                expr_value(Val1,Expr1),
                expr_value(Val2,Expr2),  !,
                Val1 < Val2.

% Expr1 > Expr2
greater_than(Expr1,Expr2) :-
                expr_value(Val1,Expr1),
                expr_value(Val2,Expr2),  !,
                Val1 > Val2.

% Expr1 =< Expr2
less_or_equal(Expr1,Expr2) :-
                expr_value(Val1,Expr1),
                expr_value(Val2,Expr2),  !,
                Val1 =< Val2.

% Expr1 >= Expr2
greater_or_equal(Expr1,Expr2) :-
                expr_value(Val1,Expr1),
                expr_value(Val2,Expr2),  !,
                Val1 >= Val2.
```

```
%%——————————————
%% Boolean-operator clauses
%%——————————————

% Relexpr1 AND Relexpr2
and(Relexpr1,Relexpr2) :-  Relexpr1, Relexpr2.

% Relexpr1 OR Relexpr2
or(Relexpr1,Relexpr2)  :-  Relexpr1 ; Relexpr2.

% NOT Relexpr1
~(Relexpr1)  :-  \+ Relexpr1.
```

APPENDIX C

A SUBSET OF QUINTUS PROLOG GRAMMAR RULES

** A Subset of Quintus Prolog Grammar Rules **

Refer to [Quintus87] for a complete listing of the rules.

Syntax of Sentences as Terms :

```
sentence          -->   clause
                   |    directive

clause            -->   non-unit clause
                   |    unit-clause

directive         -->   command
                   |    query

non-unit-clause   -->   head :- goals

unit-clause       -->   head        {where head is not otherwise a sentence }

command           -->   :- goals

query             -->   ?- goals

head              -->   term        { where term is not a number or variable }

goals             -->   goals , goals
                   |    goals ; goals
                   |    goal

goal              -->   term        {where term is not a number and is not otherwise a
                                    goals }
```

Syntax of Terms as Tokens :

```
term-read-in      -->   subterm(1200)  full-stop

subterm(N)        -->   term(M)     { where M is less than or equal to N }

term(N)           -->   op(N,fx)
                   |    op(N,fy)
                   |    op(N,fx)  subterm(N-1)
                                    { except the case '-' number }
                                    { if subterm starts with a '(', op must be followed
                                    by a space }
                   |    subterm(N-1)  op(N,xfx)  subterm(N-1)
                   |    subterm(N-1)  op(N,xfy)  subterm(N)
                   |    subterm(N)    op(N,yfx)  subterm(N-1)
                   |    subterm(N-1)  op(N,xf)
                   |    subterm(N)    op(N,yf)

term(1000)        -->   subterm(999) ,  subterm(1000)
```

```
term(0)          -->  functor(arguments)
                                { provided there is no space between functor and
                                the '(' }
                  |   ( subterm(1200) )
                  |   { subterm(1200) }
                  |   list
                  |   string
                  |   constant
                  |   variable

op(N,T)          -->  name          { where name has been declared as an operator of
                                    type T and precedence N }

arguments        -->  subterm(999)
                  |   subterm(999) ,  arguments

list             -->  []
                  |   [ listexpr ]

listexpr         -->  subterm(999)
                  |   subterm(999)  ,  listexpr
                  |   subterm(999)  |  subterm(999)

constant         -->  atom
                  |   number

number           -->  integer
                  |   float

atom             -->  name          {where  name  is not a prefix operator }

integer          -->  natural-number
                  |   - natural-number

float            -->  unsigned-float
                  |   - unsigned-float

functor          -->  name
```

<u>Syntax of Tokens as Character Strings</u> :

```
name             -->  quoted-name
                  |   word
                  |   symbol
                  |   solo-char
                  |   [ layout-char... ]
                  |   { layout-char... }

quoted-name      -->  'quoted-item...'

quoted-item      -->  char          {other than ' }
                  |   ''
```

```
word              -->   small-letter  ?alpha...

symbol            -->   symbol-char...
                                 { except in the case of a full-stop or where the first
                                 2 chars are '/*' }

natural-number    -->   digit...
                  |     base '  alphanumeric...
                                 { where each alphanumeric must be less than base ;
                                 count 'a' as 10, 'b' as 11, etc. }
                  |     zero ' char
                                 { This yields the ASCII equivalent of char }

base              -->   digit..          { must be in the range 1 .. 36 }

zero              -->   0

unsigned-float    -->   simple-float
                  |     simple-float  E  exponent

simple-float      -->   digit...  decimal-point  digit...

decimal-point     -->   .

E                 -->   E
                  |     e

exponent          -->   digit...
                  |     - digit...
                  |     + digit...

variable          -->   underline  ?alpha...
                  |     capital-letter ?alpha...

string            -->   "?string-item..."

string-item       -->   char             { other than " }
                  |     ""

space             -->   layout-char...

comment           -->   /* ?char...  */
                                 { where ?char...  must not contain '*/' }
                  |     % rest-of-line

rest-of-line      -->   newline
                  |     ?not-end-of-line ...   newline

not-end-of-line   -->   { any character  except  newline }

newline           -->   {ASCII  code 10 }
```

```
full-stop          -->   . layout-char

char               -->   layout-char
                    |    alpha
                    |    symbol-char
                    |    solo-char
                    |    punctuation-char
                    |    quote-char

layout-char        -->   {any ASCII char code up to 32 - includes space , tab , newline ,
                         and del }

alpha              -->   alphanumeric
                    |    underline

alphanumeric       -->   letter
                    |    digit

letter             -->   capital-letter
                    |    small-letter

capital-letter     -->   A | B | C | D | E | F | G | H | I | J | K | L | M | N |
                         O | P | Q | R | S | T | U | V | W | X | Y | Z

small-letter       -->   a | b | c | d | e | f | g | h | i | j | k | l | m | n |
                         o | p | q | r | s | t | u | v | w | x | y | z

digit              -->   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

symbol-char        -->   + | - | * | / | \ | ^ | < | > | =
                         | ` | ~ | : | . | ? | @ | $ | &

solo-char          -->   ; | !

punctuation-char   -->   () | [] | {} | , | |

quote-char         -->   ' | "

underline          -->   _
```

APPENDIX D

PARTIAL LISTING OF THE SOURCE CODE OF ECR

This is a partial listing of the source code of the ECR

```
% filename : mat4
% Thu Apr  4 23:04:34 CST 1991

:- [flatten].

%%----------------------------------------------------------|
%%                    TOP  PROCEDURE                        |
%%----------------------------------------------------------|

% ecr(L) --> sentences(L1), sentence(L2)
%       |    sentence(L)
sentences(G,[L1],T,A,C) :-
   link(cip_program,G),
   gl(sentence,[L2],A,B),
   flatten([L1,L2],L),
   cip_program(G,[L],T,B,C).
sentence(G,[L],T,A,B) :-
   link(cip_program,G),
   cip_program(G,[L],T,A,B).

% sentences(L) -->  sentences(L1), sentence(L2)
% sentences(L) -->  sentence(L)
sentences(G,[L1],T,A,C) :-
   link(sentences,G),
   gl(sentence,[L2],A,B),
   flatten([L1,L2],L),
   sentences(G,[L],T,B,C).
sentence(G,[L],T,A,B) :-
   link(sentences,G),
   sentences(G,[L],T,A,B).

%%-----------------------------------------------------------|
%%                           GOAL                            |
%%-----------------------------------------------------------|
gl(G,A,X,Z) :-
     ( wf_goal(G,_,X,_);
       fail_goal(G,X), !, fail), !,
       wf_goal(G,A,X,Z).
gl(G,A,X,Z) :-
    dict(C,A1,X,Y),
    % link(C,G),
    P =.. [C,G,A1,A,Y,Z], call(P),
    assertz(wf_goal(G,A,X,Z)).

gl(G,A,X,Z) :-
     ( wf_goal(G,_,X,_);
       assertz(fail_goal(G,X)) ), !, fail.

%%-----------------------------------------------------------|
%%                          RULES                            |
%%-----------------------------------------------------------|
%--------------- SYNTAX OF SENTENCES AS TERMS-------------

% sentence(L) --> claus(L)
%            | directive(L)
%
%            | grammar_rule(L)
```

```
claus(G,[L],T,X,Y) :-
    link(sentence,G),
    sentence(G,[L],T,X,Y).

directive(G,[L],T,X,Y) :-
    link(sentence,G),
    sentence(G,[L],T,X,Y).

grammar_rule(G,[L],T,X,Y) :-
    link(sentence,G),
    sentence(G,[L],T,X,Y).

% claus(L)   --> non_unit_clause(L)
%            |  unit_clause(L)
non_unit_clause(G,[L],T,X,Y) :-
    link(claus,G),
    claus(G,[L],T,X,Y).
unit_clause(G,[L],T,X,Y) :-
    link(claus,G),
    claus(G,[L],T,X,Y).

% directive(L) --> command(L)
%              |  query(L)
command(G,[L],T,X,Y) :-
    link(directive,G),
    directive(G,[L],T,X,Y).
query(G,[L],T,X,Y) :-
    link(directive,G),
    directive(G,[L],T,X,Y).

% non_unit_clause(L) --> head(L1), spc(L2), [58,45], spc(L3),
%                        goals(L4)
head(G,[L1],T,A,F) :-
    link(non_unit_clause,G),
    (gl(spc,[L2],A,B) ; A = B),
    B = [58,45|C],
    (gl(spc,[L3],C,D) ; C = D),
    gl(goals,[L4],D,E),
    flatten([L1,32,58,45,10,32,32,32,L4],L),
    non_unit_clause(G,[L],T,E,F).

% unit_clause(L) --> head(L)
% where head is not otherwise a sentence.
head(G,[L],T,X,Y) :-
    link(unit_clause,G),
    unit_clause(G,[L],T,X,Y).

% head(L) --> term_read_in(L)
%          | term(0,L1)
term(G,[0,L1],T,X,Y) :-
    link(head,G),
    head(G,[L1],T,X,Y).
term_read_in(G,[L],T,X,Y) :-
    link(head,G),
    head(G,[L],T,X,Y).
```

```
%% goals(L) --> goals(L1), spc(L2), [44], spc(L3), goals(L4)
goals(G,[L1],T,A,F) :-
    link(goals,G),
    (gl(spc,[L2],A,B) ; A = B),
    B = [44|C],
    (gl(spc,[L3],C,D) ; C = D),
    gl(goals,[L4],D,E),
    flatten([L1,44,10,32,32,32,L4],L),
    goals(G,[L],T,E,F).

                                              % [';']
%% goals(L) --> goals(L1), spc(L2), [59], spc(L3), goals(L4)
goals(G,[L1],T,A,F) :-
    link(goals,G),
    (gl(spc,[L2],A,B) ; A = B),
    B = [59|C],
    (gl(spc,[L3],C,D) ; C = D),
    gl(goals,[L4],D,E),
    flatten([L1,10,32,32,32,59,10,32,32,32,L4],L),
    goals(G,[L],T,E,F).


% goals(L) --> goal(L)
goal(G,[L],T,X,Y) :-
    link(goals,G),
    goals(G,[L],T,X,Y).

% goal(L) --> term_read_in(L)
%           | term(0,L)
%
%           | embedded_C(L)
%
%           | embedded_C(L1), full_stop(L2)
embedded_C(G,[L],T,A,B) :-
    link(goal,G),
    goal(G,[L],T,A,B).
embedded_C(G,[L1],T,A,C) :-
    link(goal,G),
    gl(full_stop,[L2],A,B),
    flatten([L1,L2],L),
    goal(G,[L],T,B,C).

term(G,[0,L],T,X,Y) :-
    link(goal,G),
    goal(G,[L],T,X,Y).
term_read_in(G,[L],T,X,Y) :-
    link(goal,G),
    goal(G,[L],T,X,Y).

                                                         % ['-->']
%% grammar_rule(L) -->  gr_head(L1), spc(L2), [45,45,62],
%                       spc(L3), gr_body(L4)
gr_head(G,[L1],T,A,F) :-
     link(grammar_rule,G),
     (gl(spc,[L2],A,B) ; A = B),
     B = [45,45,62|C],
     (gl(spc,[L3],C,D) ; C = D),
     gl(gr_body,[L4],D,E),
     flatten([L1,32,45,45,62,10,32,32,32,L4],L),
     grammar_rule(G,[L],T,E,F).
```

```
% gr_head(L) --> non_terminal(L)
non_terminal(G,[L],T,X,Y) :-
    link(gr_head,G),
    gr_head(G,[L],T,X,Y).

                                                        % [',']
%% gr_head(L) --> non_terminal(L1), spc(L2), [44],
%                 spc(L3), terminals(L4)
non_terminal(G,[L1],T,A,F) :-
    link(gr_head,G),
    (gl(spc,[L2],A,B) ; A = B),
    B = [44|C],
    (gl(spc,[L3],C,D) ; C = D),
    gl(terminals,[L4],D,E),
    flatten([L1,32,44,32,L4],L),
    gr_head(G,[L],T,E,F).

%% gr_body(L) --> gr_body(L1), spc(L2), [44],
%                 spc(L3), gr_body(L4)
gr_body(G,[L1],T,A,F) :-
    link(gr_body,G),
    (gl(spc,[L2],A,B) ; A = B),
    B = [44|C],
    (gl(spc,[L3],C,D) ; C = D),
    gl(gr_body,[L4],D,E),
    flatten([L1,44,10,32,32,32,L4],L),
    gr_body(G,[L],T,E,F).

%% gr_body(L) --> gr_body(L1), spc(L2), [59],
%                 spc(L3), gr_body(L4)
gr_body(G,[L1],T,A,F) :-
     link(gr_body,G),
     (gl(spc,[L2],A,B) ; A = B),
     B = [59|C],
     (gl(spc,[L3],C,D) ; C = D),
     gl(gr_body,[L4],D,E),
     flatten([L1,10,32,32,32,59,10,32,32,32,L4],L),
     gr_body(G,[L],T,E,F).

% gr_body(L) --> non_terminal(L)
%
%           |  terminals(L)
%
%           |  gr_condition(L)
non_terminal(G,[L],T,X,Y) :-
    link(gr_body,G),
    gr_body(G,[L],T,X,Y).
terminals(G,[L],T,X,Y) :-
    link(gr_body,G),
    gr_body(G,[L],T,X,Y).
gr_condition(G,[L],T,X,Y) :-
    link(gr_body,G),
    gr_body(G,[L],T,X,Y).

% non_terminal(L) --> term_read_in(L)
% non_terminal(L) --> term(0,L)
term_read_in(G,[L],T,X,Y) :-
    link(non_terminal,G),
    non_terminal(G,[L],T,X,Y).
```

```
term(G,[0,L],T,X,Y) :-
    link(non_terminal,G),
    non_terminal(G,[L],T,X,Y).

% terminals(L) --> list(L)
%              | string(L)
list(G,[L],T,X,Y) :-
    link(terminals,G),
    terminals(G,[L],T,X,Y).
string(G,[L],T,X,Y) :-
    link(terminals,G),
    terminals(G,[L],T,X,Y).

%% gr_condition(L) --> [123], goals(L1), [125]
terminal15(G,[123],T,A,D) :-
    link(gr_condition,G),
    gl(goals,[L1],A,B),
    B = [125|C],
    flatten([123,L1,125],L),
    gr_condition(G,[L],T,C,D).


%---------------- SYNTAX OF TERMS AS TOKENS --------------|

% term_read_in(L) --> subterm(1200,L1), full_stop(L2)
subterm(G,[1200,L1],T,X,Z) :-
    link(term_read_in,G),
    gl(full_stop,[L2],X,Y),
    flatten([L1,L2],L),
    term_read_in(G,[L],T,Y,Z).

% subterm(N,L) --> term(M,L), {M=<N}
term(G,[M,L],T,X,Y) :-
    link(subterm,G),
    subterm(G,[N,L],T,X,Y),
    integer(M), integer(N),
    M =< N.

                                  %['('] [')']
%% term(0,L) --> functor(L1), [40], arguments(L2), [41]


funct(G,[L1],T,A,E) :-
    link(term,G),
    A = [40|B],
    gl(arguments,[L2],B,C),
    C = [41|D],
    flatten([L1,40,L2,41],L),
    term(G,[0,L],T,D,E).

%% term(0,L) --> [40], subterm(1200,L1), [41]
terminal14(G,_,T,A,D) :-
    link(term,G),
    gl(subterm,[1200,L1],A,B),
    B = [41|C],
    flatten([40,L1,41],L),
    term(G,[0,L],T,C,D).
```

```
                  % ['{']                        ['}']
%% term(0,L) --> [123], subterm(1200,L1), [125]
terminal13(G,_,T,A,D) :-
    link(term,G),
    gl(subterm,[1200,L1],A,B),
    B = [125|C],
    flatten([123,L1,125],L),
    term(G,[0,L],T,C,D).

% term(0,L) --> list(L)
%            | string(L)
%
%            | constant(L)
%
%            | variable(L)
list(G,[L],T,X,Y) :-
    link(term,G),
    term(G,[0,L],T,X,Y).
string(G,[L],T,X,Y) :-
    link(term,G),
    term(G,[0,L],T,X,Y).
constant(G,[L],T,X,Y) :-
    link(term,G),
    term(G,[0,L],T,X,Y).
variable(G,[L],T,X,Y) :-
    link(term,G),
    term(G,[0,L],T,X,Y).

% term(N,L) --> op(N,fx,L)
op(G,[N,fx,L],T,X,Y) :-
    link(term,G),
    term(G,[N,L],T,X,Y).

% term(N,L) --> op(N,fy,L)
op(G,[N,fy,L],T,X,Y) :-
    link(term,G),
    term(G,[N,L],T,X,Y).

% term(N,L) --> op(N,fx,L1), {M is N-1}, subterm(M,L2)
op(G,[N,fx,L1],T,X,Z) :-
    link(term,G),
    integer(N),
    M is N-1,
    gl(subterm,[M,L2],X,Y),
    flatten([L1,L2],L),
    term(G,[N,L],T,Y,Z).

% term(N,L) --> op(N,fy,L1), subterm(N,L2)
%                  {flatten([L1,L2],L)}.
%                           if subterm starts
%                           with a '(',  op must
%                           be followed by a space
op(G,[N,fy,L1],T,X,Z) :-
    link(term,G),
    gl(subterm,[N,L2],X,Y),
    flatten([L1,L2],L),
    term(G,[N,L],T,Y,Z).

% term(N,L) --> {M is N-1}, subterm(M,L1), op(N,xfx,L2),
% subterm(M,L3), flatten([L1,L2,L3],L)}
```

```
subterm(G,[M,L1],T,X,V) :-
    link(term,G),
    gl(op,[N,xfx,L2],X,Y),
    gl(subterm,[M,L3],Y,Z),
    integer(M),
    integer(N),
    M1 is N-1,
    M1 =:= M,
    flatten([L1,L2,L3],L),
    term(G,[N,L],T,Z,V).

% term(N,L) --> {M is N-1}, subterm(M,L1), op(N,xfy,L2),
%             subterm(N,L3)
subterm(G,[M,L1],T,X,V) :-
    link(term,G),
    gl(op,[N,xfy,L2],X,Y),
    gl(subterm,[N,L3],Y,Z),
    integer(M),
    integer(N),
    M1 is N-1,
    M1 =:= M,
    flatten([L1,L2,L3],L),
    term(G,[N,L],T,Z,V).


% term(N,L) --> subterm(N,L1), op(N,yfx,L2), {M is N-1},
% subterm(M,L3)
subterm(G,[N,L1],T,X,V) :-
    link(term,G),
    gl(op,[N,yfx,L2],X,Y),
    integer(N),
    M is N-1,
    gl(subterm,[M,L3],Y,Z),
    flatten([L1,L2,L3],L),
    term(G,[N,L],T,Z,V).

% term(N,L) --> {M is N-1}, subterm(M,L1), op(N,xf,L2)
subterm(G,[M,L1],T,X,Z) :-
    link(term,G),
    gl(op,[N,xf,L2],X,Y),
    integer(M),
    integer(N),
    M1 is N-1,
    M1 =:= M,
    flatten([L1,L2],L),
    term(G,[N,L],T,Y,Z).

% term(N,L) --> subterm(N,L1), op(N,yf,L2)
subterm(G,[N,L1],T,X,Z) :-
    link(term,G),
    gl(op,[N,yf,L2],X,Y),
    flatten([L1,L2],L),
    term(G,[N,L],T,Y,Z).

                                                   %[',']
%% term(1000,L) --> subterm(999,L1), spc(L2), [44],
%                   spc(L3), subterm(1000,L4)
subterm(G,[999,L1],T,A,F) :-
    link(term,G),
    (gl(spc,[L2],A,B) ; A = B),
```

```
    B = [44|C],
    (gl(spc,[L3],C,D) ; C = D),
    gl(subterm,[1000,L4],D,E),
    flatten([L1,44,L4],L),
    term(G,[1000,L],T,E,F).

% arguments(L) --> subterm(999,L)
subterm(G,[999,L],T,X,Y) :-
    link(arguments,G),
    arguments(G,[L],T,X,Y).

                                                      %[',']
% arguments(L) --> subterm(999,L1), spc(L2), [44],
%                  spc(L3), arguments(L4)
subterm(G,[999,L1],T,A,F) :-
    link(arguments,G),
    (gl(spc,[L2],A,B) ; A = B),
    B = [44|C],
    (gl(spc,[L3],C,D) ; C = D),
    gl(arguments,[L4],D,E),
    flatten([L1,44,L4],L),
    arguments(G,[L],T,E,F).

            %['[']  [']']
% list(L) --> [91], [93]
terminal12(G,_,T,A,C) :-
    link(list,G),
    A = [93|B],
    flatten([91,93],L),
    list(G,[L],T,B,C).

            %['[']                  [']']
% list(L) --> [91], listexpr(L1), [93]
terminal12(G,_,_,A,D) :-
    link(list,G),
    gl(listexpr,[L1],A,B),
    B = [93|C],
    flatten([91,L1,93],L),
    list(G,[L],_,C,D).

% listexpr(L) --> subterm(999,L)
subterm(G,[999,L],T,X,Y) :-
    link(listexpr,G),
    listexpr(G,[L],T,X,Y).

% listexpr(L) --> subterm(999,L1), spc(L2), [44],
%                 spc(L3), listexpr(L4)
subterm(G,[999,L1],T,A,F) :-
    link(listexpr,G),
    (gl(spc,[L2],A,B) ; A = B),
    B = [44|C],
    (gl(spc,[L3],C,D) ; C = D),
    gl(listexpr,[L4],D,E),
    flatten([L1,44,L4],L),
    listexpr(G,[L],T,E,F).

                                          % ['|']
% listexpr(L) --> subterm(999,L1), [124], subterm(999,L2)
subterm(G,[999,L1],T,A,D) :-
    link(listexpr,G),
```

```
    A = [124|B],
    gl(subterm,[999,L2],B,C),
    flatten([L1,124,L2],L),
    listexpr(G,[L],T,C,D).

% constant(L) --> atom(L)
%             | number(L)
atom(G,[L],T,X,Y) :-
    link(constant,G),
    constant(G,[L],T,X,Y).
number(G,[L],T,X,Y) :-
    link(constant,G),
    constant(G,[L],T,X,Y).

% number(L) --> integer(L)
%           | float(L)
integer(G,[L],T,X,Y) :-
    link(number,G),
    number(G,[L],T,X,Y).
float(G,[L],T,X,Y) :-
    link(number,G),
    number(G,[L],T,X,Y).

% atom(L) --> name1(L)
name1(G,[L],T,X,Y) :-
    link(atom,G),
    atom(G,[L],T,X,Y).

% functor(L) --> name1(L)
name1(G,[L],T,X,Y) :-
    link(funct,G),
    funct(G,[L],T,X,Y).

% integer(L) --> natural_number(L)
natural_number(G,[L],T,X,Y) :-
    link(integer,G),
    integer(G,[L],T,X,Y).

% integer(L) --> [45], natural_number(L1),
%                {flatten([45,L1],L)}
terminal11(G,_,T,A,C) :-
    link(integer,G),
    gl(natural_number,[L1],A,B),
    flatten([45,L1],L),
    integer(G,[L],T,B,C).

% float(L) --> unsigned_float(L)
unsigned_float(G,[L],T,X,Y) :-
    link(float,G),
    float(G,[L],T,X,Y).

% float(L) --> [45], unsigned_float(L1),
% {flatten([45,L1],L)}
terminal10(G,_,T,A,C) :-
    link(float,G),
    gl(float,[L1],A,B),
    flatten([45,L1],L),
    unsigned_float(G,[L],T,B,C).
```

```
% filename: crecg
% Tue Apr 23 10:51:05 CDT 1991

:- ['link_rtc_ttl.pl'].

%%--------------------------------------------------------|
%%                        RULES                           |
%%--------------------------------------------------------|

% declaration --> type_spec, c_spc, init_decl_list, ';',
%                 c_spc
type_spec(G,[L1],T,A,F) :-
   link(declaration,G),
   gl(c_spc,[L2],A,B),
   gl(init_decl_list,[L3],B,C),
   C = [59|D],
   (
     gl(c_spc,[L4],D,E)
     ;
     D = E, L4 = []
   ),
   flatten([L1,L2,L3,59,L4],L),
   declaration(G,[L],T,E,F).


% init_decl_list --> declarator, c_spc
% init_decl_list --> declarator, c_spc, ',', c_spc,
% init_decl_list, c_spc
declarator(G,[L1],T,A,C) :-
   link(init_decl_list,G),
   (
    gl(c_spc,[L2],A,B)
    ;
    A = B, L2 =[]
   ),
   flatten([L1,L2],L),
   init_decl_list(G,[L],T,B,C).
declarator(G,[L1],T,A,H) :-
   link(init_decl_list,G),
   (
    gl(c_spc,[L2],A,B)
    ;
    A = B, L2 =[]
   ),
   B = [44|C],
   ( gl(c_spc,[L3],C,D)
    ;
    C = D, L3 =[] ),
   gl(init_decl_list,[L4],D,E),
   ( gl(c_spc,[L5],E,F)
    ;
    E = F, L5 =[] ),
   flatten([L1,L2,44,L3,L4,L5],L),
   init_decl_list(G,[L],T,F,H).

% declarator --> identifier
identifier(G,[L],T,A,B) :-
   link(declarator,G),
   declarator(G,[L],T,A,B).
```

```
% declarations --> declaration, c_spc
% declarations --> declarations, c_spc, declaration, c_spc
declaration(G,[L1],T,A,C) :-
   link(declarations,G),
   ( gl(c_spc,[L2],A,B)
    ;
    A = B, L2 =[] ),
   flatten([L1,L2],L),
   declarations(G,[L],T,B,C).

declarations(G,[L1],T,A,E) :-
   link(declarations,G),
   ( gl(c_spc,[L2],A,B)
    ;
    A = B, L2 =[] ),
   gl(declaration,[L3],B,C),
   ( gl(c_spc,[L4],C,D)
    ;
    C = D, L4 =[] ),
   flatten([L1,L2,L3,L4],L),
   declarations(G,[L],T,D,E).

% c_space --> c_layout_chars
c_layout_chars(G,[L],T,X,Y) :-
    link(c_space,G),
    c_space(G,[L],T,X,Y).

% c_spc --> c_space
%         |    []
c_space(G,[L],T,X,Y) :-
    link(c_spc,G),
    c_spc(G,[L],T,X,Y).


% c_layout_chars --> c_layout_char, c_layout_chars
%                  |  c_layout_char
c_layout_char(G,[L1],T,X,Z) :-
    link(c_layout_chars,G),
    gl(c_layout_chars,[L2],X,Y),
    flatten([L1,L2],L),
    c_layout_chars(G,[L],T,Y,Z).
c_layout_char(G,[L],T,X,Y) :-
    link(c_layout_chars,G),
    c_layout_chars(G,[L],T,X,Y).


% identifier -->
%     (c_letter ; under_score_char),
%     (true ; dgts),
%     (identifier ; true)
c_letter(G,[L1],T,A,D) :-
   link(identifier,G),
   ( A = B, L2 =[] ;
     gl(dgts,[L2],A,B)),
   (gl(identifier,[L3],B,C)
    ;
    (B=C, L3 = [])),
   flatten([L1,L2,L3],L),
   identifier(G,[L],T,C,D).
```

```
under_score_char(G,[95],T,A,D) :-
   link(identifier,G),
   ( A = B, L2 =[]
     ;
     gl(dgts,[L2],A,B)
   ),
   ( gl(identifier,[L3],B,C)
     ;
     (B=C, L3 = [])
   ),
   flatten([95,L2,L3],L),
   identifier(G,[L],T,C,D).

% dgts --> dgts, c_digit
dgts(G,[L1],T,A,C) :-
   link(dgts,G),
   gl(c_digit,[L2],A,B),
   flatten([L1,L2],L),
   dgts(G,[L],T,B,C).

% dgts --> c_digit
c_digit(G,[L],T,A,B) :-
   link(dgts,G),
   dgts(G,[L],T,A,B).


% c_constant --> float_const
%              |   int_const
float_const(G,[L],T,A,B) :-
   link(c_constant,G),
   c_constant(G,[L],T,A,B).
int_const(G,[L],T,A,B) :-
   link(c_constant,G),
   c_constant(G,[L],T,A,B).

% float_const --> float_const_1
float_const_1(G,[L],T,A,B) :-
   link(float_const,G),
   float_const(G,[L],T,A,B).

% float_const_1 --> fract_const, (true ; exponent_part)
fract_const(G,[L1],T,A,C) :-
   link(float_const_1,G),
   ((A=B, L2 = []) ;
    (gl(exponent_part,[L2],A,B))
   ),
   flatten([L1,L2],L),
   float_const_1(G,[L],T,B,C).

% float_const_1 --> dgts, exponent_part
dgts(G,[L1],T,A,C) :-
   link(float_const_1,G),
   gl(exponent_part,[L2],A,B),
   flatten([L1,L2],L),
   float_const_1(G,[L],T,B,C).

% fract_const --> dgts, '.', dgts
% fract_const --> dgts, '.'
dgts(G,[L1],T,A,D) :-
   link(fract_const,G),
```

```
   A = [46|B],
   gl(dgts,[L2],B,C),
   flatten([L1,46,L2],L),
   fract_const(G,[L],T,C,D).
dgts(G,[L1],T,A,C) :-
   link(fract_const,G),
   A = [46|B],
   flatten([L1,46],L),
   fract_const(G,[L],T,B,C).

% exponent_part -->
%     ('e' ; 'E'),
%     (true ; '+' ; '-'),
%     dgts
termnl5(G,[L1],T,A,D) :-
    link(exponent_part,G),
    (L1 = 101 ; L1 = 69),
    ((A=B, L2=[]) ; (A=[43|B], L2=[43]) ;
     (A=[45|B], L2=[45])
    ),
    gl(dgts,[L3],B,C),
    flatten([L1,L2,L3],L),
    exponent_part(G,[L],T,C,D).

% int_const --> nonzero_dgt, dgts
% int_const --> nonzero_dgt
nonzero_dgt(G,[L],T,A,B) :-
    link(int_const,G),
    int_const(G,[L],T,A,B).
nonzero_dgt(G,[L1],T,A,C) :-
    link(int_const,G),
    gl(dgts,[L2],A,B),
    flatten([L1,L2],L),
    int_const(G,[L],T,B,C).

% char_const --> '''', c_chars, ''''
termnl6(G,[96],T,A,D) :-
   link(char_const,G),
   gl(c_chars,[L1],A,B),
   B = [96|C],
   flatten([96,L1,96],L),
   char_const(G,[L],T,C,D).

% expression --> primary_expr
% expression --> expression, c_spc, assign_opr, c_spc,
%                expression
% expression --> expression, c_spc, bin_opr, c_spc,
%                expression
primary_expr(G,[L],T,A,B) :-
   link(expression,G),
   expression(G,[L],T,A,B).
expression(G,[L1],T,A,F) :-
   link(expression,G),
   ( gl(c_spc,[L2],A,B)
    ;
    A = B, L2 =[] ),
   gl(assign_opr,[L3],B,C),
   ( gl(c_spc,[L4],C,D)
    ;
    C = D, L4 =[] ),
```

```
   gl(expression,[L5],D,E),
   flatten([L1,L2,L3,L4,L5],L),
   expression(G,[L],T,E,F).
expression(G,[L1],T,A,F) :-
   link(expression,G),
   ( gl(c_spc,[L2],A,B)
    ;
    A = B, L2 =[] ),
   gl(bin_opr,L3,B,C),
   ( gl(c_spc,[L4],C,D)
    ;
    C = D, L4 =[] ),
   gl(expression,[L5],D,E),
   flatten([L1,L2,L3,L4,L5],L),
   expression(G,[L],T,E,F).

% primary_expr --> identifier
%                |  c_constant
%
%               | string_literal
%
%               | '(', expression, ')'
identifier(G,[L],T,A,B) :-
    link(primary_expr,G),
    primary_expr(G,[L],T,A,B).
c_constant(G,[L],T,A,B) :-
    link(primary_expr,G),
    primary_expr(G,[L],T,A,B).
termnl8(G,[40],T,A,D) :-
    link(primary_expr,G),
    gl(expression,[L1],A,B),
    B = [41|C],
    flatten([40,L1,41],L),
    primary_expr(G,[L],T,C,D).

% embedded_C(L) --> compound_st(L)
compound_st(G,[L],T,X,Y) :-
    link(embedded_C,G),
    embedded_C(G,[L],T,X,Y).

% statement -->
%           | compound_st
%
%           | expression_st
%
%           | selection_st
%
%           | iteration_st
compound_st(G,[L],T,A,B) :-
   link(statement,G),
   statement(G,[L],T,A,B).
expression_st(G,[L],T,A,B) :-
   link(statement,G),
   statement(G,[L],T,A,B).

selection_st(G,[L],T,A,B) :-
   link(statement,G),
   statement(G,[L],T,A,B).

iteration_st(G,[L],T,A,B) :-
```

```
   link(statement,G),
   statement(G,[L],T,A,B).

% compound_st -->
%   '{', (true ; c_spc, declarations), c_spc,
%        (true ; stmts), c_spc,
%   '}'
termnl9(G,[123],T,A,I) :-
   link(compound_st,G),
   ( A=C, L1=[], L2 = []
     ;
     ( gl(c_spc,[L1],A,B)
       ;
       A = B, L1 =[]
     ),
     gl(declarations,[L2],B,C)
   ),
   ( gl(c_spc,[L3],C,D)
    ;
    C = D, L3 =[]
   ),
   ( D=E, L4=[] ;
     gl(stmts,[L4],D,E)
   ),
   ( gl(c_spc,[L5],E,F)
    ;
    E = F, L5 =[]
   ),
   F = [125|H],
   flatten([123,L1,L2,L3,L4,L5,125],L),
   compound_st(G,[L],T,H,I).


% stmts --> statement
% stmts --> stmts,c_spc,statement
statement(G,[L],T,A,B) :-
   link(stmts,G),
   stmts(G,[L],T,A,B).
stmts(G,[L1],T,A,D) :-
   link(stmts,G),
   ( gl(c_spc,[L2],A,B)
    ;
    A = B, L2 =[]
   ),
   gl(statement,[L3],B,C),
   flatten([L1,L2,L3],L),
   stmts(G,[L],T,C,D).


% expression_st --> (true ; expression), c_spc, ';'
termnl10(G,[59],T,A,B) :-
    link(expression_st,G),
    expression_st(G,[59],T,A,B).
c_spc(G,[L1],T,A,C) :-
    link(expression_st,G),
    A = [59|B],
    flatten([L1,59],L),
    expression_st(G,[L],T,B,C).
expression(G,[L1],T,A,D) :-
   link(expression_st,G),
```

```
   ( gl(c_spc,[L2],A,B)
    ;
    A = B, L2 =[]
   ),
   B = [59|C],
   flatten([L1,L2,59],L),
   expression_st(G,[L],T,C,D).

% selection_st -->
%    'if', '(', c_spc, expression, c_spc, ')', c_spc, %
statement
% selection_st -->
%    'if', '(', c_spc, expression, c_spc, ')',
%     c_spc, statement, c_spc, 'else', c_spc, statement
%
%%***eliminating ambiguity for selection_st rules:
%
%% selection_st --> matched_st
%%                |  unmatched_st
matched_st(G,[L],T,A,B) :-
   link(selection_st,G),
   selection_st(G,[L],T,A,B).
unmatched_st(G,[L],T,A,B) :-
   link(selection_st,G),
   selection_st(G,[L],T,A,B).


% matched_st --> 'if', c_spc, '(', c_spc, expression, c_spc,
%                 ')', c_spc, matched_st, c_spc, 'else',c_spc,
%                 matched_st
%             |  compound_st
%
%             |  expression_st
%
%             |  iteration_st
% unmatched_st --> 'if', c_spc, '(', c_spc, expression,
%                   c_spc, ')',c_spc, unmatched_st
%             | 'if', c_spc, '(', c_spc, expression, c_spc,
%               ')', c_spc, matched_st, c_spc, 'else', c_spc,
%               unmatched_st
%             |  compound_st
%
%             |  expression_st
%
%             |  iteration_st
termnl11(G,[105,102,40],T,A,P) :-
   link(matched_st,G),
   ( gl(c_spc,[L1],A,B)
    ;
    A = B, L1 =[]
   ),
   gl(expression,[L2],B,C),
   ( gl(c_spc,[L3],C,D)
    ;
    C = D, L3 =[]
   ),
   D = [41|E],
   ( gl(c_spc,[L4],E,F)
    ;
    E = F, L4 =[]
```

```
   ),
   gl(matched_st,[L5],F,H),
   ( gl(c_spc,[L6],H,I)
    ;
    H = I, L6 =[]
   ),
   I = [101,108,115,101|J],
   ( gl(c_spc,[L7],J,K)
    ;
    J = K, L7 =[]
   ),
   gl(matched_st,[L8],K,M),
   ( gl(c_spc,[L9],M,N)
    ;
    M = N, L9 =[]
   ),
   flatten([105,102,40,L1,L2,L3,41,L4,L5,L6,101,108,115,101,
            L7,L8,L9],L),
   matched_st(G,[L],T,N,P).

compound_st(G,[L],T,A,B) :-
   link(matched_st,G),
   matched_st(G,[L],T,A,B).

expression_st(G,[L],T,A,B) :-
   link(matched_st,G),
   matched_st(G,[L],T,A,B).

iteration_st(G,[L],T,A,B) :-
   link(matched_st,G),
   matched_st(G,[L],T,A,B).

termnl11(G,[105,102,40],T,A,J) :-
   link(unmatched_st,G),
   ( gl(c_spc,[L1],A,B)
    ;
    A = B, L1 =[]
   ),
   gl(expression,[L2],B,C),
   ( gl(c_spc,[L3],C,D)
    ;
    C = D, L3 =[]
   ),
   D = [41|E],
   ( gl(c_spc,[L4],E,F)
    ;
    E = F, L4 =[]
   ),
   gl(unmatched_st,[L5],F,H),
   ( gl(c_spc,[L6],H,I)
    ;
    H = I, L6 =[]
   ),
   flatten([105,102,40,L1,L2,L3,41,L4,L5,L6],L),
   unmatched_st(G,[L],T,I,J).

termnl11(G,[105,102,40],T,A,P) :-
   link(unmatched_st,G),
   ( gl(c_spc,[L1],A,B)
    ;
```

```
     A = B, L1 =[]
    ),
    gl(expression,[L2],B,C),
    ( gl(c_spc,[L3],C,D)
     ;
     C = D, L3 =[]
    ),
    D = [41|E],
    ( gl(c_spc,[L4],E,F)
     ;
     E = F, L4 =[]
    ),
    gl(matched_st,[L5],F,H),
    ( gl(c_spc,[L6],H,I)
     ;
     H = I, L6 =[]
    ),
    I = [101,108,115,101|J],
    ( gl(c_spc,[L7],J,K)
     ;
     J = K, L7 =[]
    ),
    gl(unmatched_st,[L8],K,M),
    ( gl(c_spc,[L9],M,N)
     ;
     M = N, L9 =[]
    ),
    flatten([105,102,40,L1,L2,L3,41,L4,L5,L6,101,108,115,101,
             L7,L8,L9],L),
    unmatched_st(G,[L],T,N,P).

compound_st(G,[L],T,A,B) :-
    link(unmatched_st,G),
    unmatched_st(G,[L],T,A,B).

expression_st(G,[L],T,A,B) :-
    link(unmatched_st,G),
    unmatched_st(G,[L],T,A,B).

iteration_st(G,[L],T,A,B) :-
    link(unmatched_st,G),
    unmatched_st(G,[L],T,A,B).


% iteration_st -->
%    'while', '(', c_spc, expression, c_spc,')',c_spc,
%    statement
% iteration_st -->
%    'do', c_spc, statement, c_spc, 'while', '(',
%    c_spc, expression, c_spc, ')',c_spc,';'
% iteration_st -->
%    'for', '(', c_spc,
%               (true ; expression, c_spc), ';',
%                c_spc,
%               (true ; expression, c_spc), ';',
%                c_spc,
%               (true ; expression, c_spc),
%           ')', c_spc, statement
termnl12(G,[119,104,105,108,101,40],T,A,I) :-
    link(iteration_st,G),
```

```
   ( gl(c_spc,[L1],A,B)
    ;
    A = B, L1 =[]
   ),
   gl(expression,[L2],B,C),
   ( gl(c_spc,[L3],C,D)
    ;
    C = D, L3 =[]
   ),
   D = [41|E],
   ( gl(c_spc,[L4],E,F)
    ;
    E = F, L4 =[]
   ),
   gl(statement,[L5],F,H),
   flatten([119,104,105,108,101,40,L1,L2,L3,41,L4,L5],L),
   iteration_st(G,[L],T,H,I).

termnl13(G,[100,111],T,A,M) :-
   link(iteration_st,G),
   gl(c_spc,[L1],A,B),
   gl(statement,[L2],B,C),
   ( gl(c_spc,[L3],C,D)
    ;
    C = D, L3 =[]
   ),
   D = [119,104,105,108,101,40|E],
   ( gl(c_spc,[L4],E,F)
    ;
    E = F, L4 =[]
   ),
   gl(expression,[L5],F,H),
   ( gl(c_spc,[L6],H,I)
    ;
    H = I, L6 =[]
   ),
   I = [41,59|J],
   ( gl(c_spc,[L7],J,K)
    ;
    J = K, L7 =[]
   ),
   flatten([100,111,L1,L2,L3,119,104,105,108,101,40,
            L4,L5,L6,41,59,L7],L),
   iteration_st(G,[L],T,K,M).

termnl14(G,[102,111,114,40],T,A,U) :-
   link(iteration_st,G),
   ( gl(c_spc,[L1],A,B)
    ;
    A=B, L1 =[]
   ),
   ( gl(expression,[L2],B,C),
     ( gl(c_spc,[L3],C,D)
       ;
       C=D, L3 =[]
     )
     ;
     B=D, L2=[], L3=[]
   ),
   D=[59|E],
```

```
    ( gl(c_spc,[L4],E,F)
     ;
     E=F, L4 =[]
    ),
    ( gl(expression,[L5],F,H),
      ( gl(c_spc,[L6],H,I)
        ;
        H=I, L6 =[]
      )
      ;
      F=I, L5=[], L6=[]),
    I=[59|J],
    ( gl(c_spc,[L7],J,K)
     ;
     J=K, L7 =[]
    ),
    ( gl(expression,[L8],K,M),
      ( gl(c_spc,[L9],M,N)
        ;
        M=N, L9 =[]
      )
      ;
      K=N, L8=[], L9=[]
    ),
    N=[41|P],
    ( gl(c_spc,[L10],P,Q)
     ;
     P=Q, L10 =[]
    ),
    gl(statement,[L11],Q,R),
    ( gl(c_spc,[L12],R,S)
     ;
     R=S, L12 =[]
    ),
    flatten([102,111,114,40,L1,L2,L3,59,L4,L5,L6,59,
             L7,L8,L9,41,L10,L11,L12],L),
    iteration_st(G,[L],T,S,U).

termnl16(G,[102,111,114,40,59,59,41],T,A,C) :-
    link(iteration_st,G),
    gl(statement,[L1],A,B),
    flatten([102,111,114,40,L1,59,59,41,L1],L),
    iteration_st(G,[L],T,B,C).
```

```
%-----------------------------------------------------------|
%                        DICTIONARY                         |
%-----------------------------------------------------------|

% not_end_of_line(A) --> {any character except newline}
dict(not_end_of_line,[A],[A|X],X) :-  A \== 10.

% zero(A) --> [48]
dict(zero,[48],[48|X],X).

% decimal_point(L) --> [46]
dict(decimal_point,[46],[46|X],X).

% e_symbol(L) --> [69]
%             |  [101]
dict(e_symbol,[69],[69|X],X).
dict(e_symbol,[101],[101|X],X).

% newline(L) --> [10].
dict(newline,[10],[10|X],X).

% layout_char(A) -->  [0] | [9] | [10] | [27] | [32] | [127]
dict(layout_char,[A],[A|X],X) :-
   (A == 0 ; A == 9 ; A == 10 ;
   A == 27 ; A == 32 ; A == 127).

% capital_letter(A) -->
%                     [65] | [66] | [67] | [68] | [69] | [70]
%
%                   | [71] | [72] | [73] | [74] | [75] | [76]
%
%                   | [77] | [78] | [79] | [80] | [81] | [82]
%
%                   | [83] | [84] | [85] | [86] | [87] | [88]
%
%                   | [89] | [90]
dict(capital_letter,[A],[A|X],X) :-
    A >= 65, A =< 90.

% small_letter(A) -->
%                 [97] | [98] | [99] | [100] | [101] | [102]
%               | [103] | [104] | [105] | [106] | [107] | [108]
%
%               | [109] | [110] | [111] | [112] | [113] | [114]
%
%               | [115] | [116] | [117] | [118] | [119] | [120]
%
%               | [121] | [122]
dict(small_letter,[A],[A|X],X) :-
    A >= 97, A =< 122.

% digit(A) --> [48] | [49] | [50] | [51] | [52] | [53] | [54]
%            | [55] | [56] | [57]
dict(digit,[A],[A|X],X) :-
    A >= 48, A =< 57.

% symbol_char(A) -->  [36] | [37] | [38]
%                | [42] | [43] | [45] | [46] | [47] | [58]
%
%                | [60] | [61] | [62] | [63] | [64]
```

```
%               | [92] | [94] | [96]
%
%               | [126]
dict(symbol_char,[A],[A|X],X) :-
    ( (A >= 36, A =< 38) ;
     A == 42 ; A == 43 ;
     (A >= 45, A =< 47) ;
     A == 58 ;
     (A >= 60, A =< 64) ;
     A == 92 ; A == 94 ;
     A == 96 ; A == 126 ).

% solo_char(L) --> [59] | [33]
dict(solo_char,[A],[A|X],X) :-
    (A == 59 ; A == 33).

% punctuation_char(A) --> [40] | [41] | [44] | [91] | [93]
%                   | [123]| [124] | [125]
dict(punctuation_char,[A],[A|X],X) :-
     ( A == 40 ; A == 41 ; A == 44 ;
      A == 91 ; A == 93 ; A == 123 ;
      A == 124 ; A == 125).

% quote_char(A) --> [39] | [34]
dict(quote_char,[A],[A|X],X) :-
    (A == 39 ; A == 34).

% underline(L) --> [95]
dict(underline,[95],[95|X],X).
```

```
%%-----------------------------------------------------
%%                 TERMINATE  CLAUSES
%%-----------------------------------------------------

sentence(sentence,T,T,X,X).
sentences(sentences,T,T,X,X).
claus(claus,T,T,X,X).
directive(directive,T,T,X,X).
non_unit_clause(non_unit_clause,T,T,X,X).
unit_clause(unit_clause,T,T,X,X).
command(command,T,T,X,X).
query(query,T,T,X,X).
head(head,T,T,X,X).
goals(goals,T,T,X,X).
goal(goal,T,T,X,X).
grammar_rule(grammar_rule,T,T,X,X).
gr_head(gr_head,T,T,X,X).
gr_body(gr_body,T,T,X,X).
non_terminal(non_terminal,T,T,X,X).
terminals(terminals,T,T,X,X).
gr_condition(gr_condition,T,T,X,X).
term_read_in(term_read_in,T,T,X,X).
arguments(arguments,T,T,X,X).
list(list,T,T,X,X).
listexpr(listexpr,T,T,X,X).
constant(constant,T,T,X,X).
number(number,T,T,X,X).
atom(atom,T,T,X,X).
funct(funct,T,T,X,X).
integer(integer,T,T,X,X).
float(float,T,T,X,X).
token(token,T,T,X,X).
name1(name1,T,T,X,X).
quoted_name(quoted_name,T,T,X,X).
quoted_items(quoted_items,T,T,X,X).
quoted_item(quoted_item,T,T,X,X).
word(word,T,T,X,X).
alphas(alphas,T,T,X,X).
symbol(symbol,T,T,X,X).
symbol_chars(symbol_chars,T,T,X,X).
natural_number(natural_number,T,T,X,X).
alphanumerics(alphanumerics,T,T,X,X).
base(base,T,T,X,X).
zero(zero,T,T,X,X).
unsigned_float(unsigned_float,T,T,X,X).
simple_float(simple_float,T,T,X,X).
decimal_point(decimal_point,T,T,X,X).
e_symbol(e_symbol,T,T,X,X).
exponent(exponent,T,T,X,X).
variable(variable,T,T,X,X).
string(string,T,T,X,X).
string_items(string_items,T,T,X,X).
string_item(string_item,T,T,X,X).
space(space,T,T,X,X).
spc(spc,T,T,X,X).
comment(comment,T,T,X,X).
rest_of_line(rest_of_line,T,T,X,X).
not_end_of_lines(not_end_of_lines,T,T,X,X).
not_end_of_line(not_end_of_line,T,T,X,X).
newline(newline,T,T,X,X).
```

```
full_stop(full_stop,T,T,X,X).
chars(chars,T,T,X,X).
char(char,T,T,X,X).
layout_chars(layout_chars,T,T,X,X).
layout_char(layout_char,T,T,X,X).
alpha(alpha,T,T,X,X).
alphanumeric(alphanumeric,T,T,X,X).
letter(letter,T,T,X,X).
capital_letter(capital_letter,T,T,X,X).
small_letter(small_letter,T,T,X,X).
digits(digits,T,T,X,X).
digit(digit,T,T,X,X).
symbol_char(symbol_char,T,T,X,X).
solo_char(solo_char,T,T,X,X).
punctuation_char(punctuation_char,T,T,X,X).
quote_char(quote_char,T,T,X,X).
underline(underline,T,T,X,X).
```

```
%----------------------------------------------------------------
%                          LINK
%----------------------------------------------------------------
link(alpha,alpha).
link(alpha,alphas).
link(alpha,char).
link(alpha,chars).
link(alphanumeric,alpha).
link(alphanumeric,alphanumeric).
link(alphanumeric,alphanumerics).
link(alphanumeric,alphas).
link(alphanumeric,char).
link(alphanumeric,chars).
link(alphanumeric,quoted_item).
link(alphanumeric,quoted_items).
link(alphanumeric,string_items).
link(alpha,quoted_item).
link(alpha,quoted_items).
link(alphas,alphas).
link(alpha,string_item).
link(alpha,string_items).
link(arguments,arguments).
link(assign_expr,assign_expr).
link(assign_expr,init_decl_list_3).
link(atom,arguments).
link(atom,atom).
link(atom,cip_program).
link(atom,claus).
link(atom,constant).
link(atom,goal).
link(atom,goals).
link(atom,grammar_rule).
link(atom,gr_body).
link(atom,gr_head).
link(atom,head).
link(atom,listexpr).
link(atom,non_terminal).
link(atom,non_unit_clause).
link(atom,sentence).
link(atom,sentences).
link(atom,subterm).
link(atom,term).
link(atom,term_read_in).
link(atom,unit_clause).
link(base,arguments).
link(base,base).
link(base,cip_program).
link(base,claus).
link(base,constant).
link(base,goal).
link(base,goals).
link(base,grammar_rule).
link(base,gr_body).
link(base,gr_head).
link(base,head).
link(base,integer).
link(base,listexpr).
link(base,natural_number).
link(base,non_terminal).
link(base,non_unit_clause).
```

```
link(base,number).
link(base,sentence).
link(base,sentences).
link(base,subterm).
link(base,term).
link(base,term_read_in).
link(base,token).
link(base,unit_clause).
link(capital_letter,alpha).
link(capital_letter,alphanumeric).
link(capital_letter,alphanumerics).
link(capital_letter,alphas).
link(capital_letter,arguments).
link(capital_letter,capital_letter).
link(capital_letter,char).
link(capital_letter,chars).
link(capital_letter,cip_program).
link(capital_letter,claus).
link(capital_letter,goal).
link(capital_letter,goals).
link(capital_letter,grammar_rule).
link(capital_letter,gr_body).
link(capital_letter,gr_head).
link(capital_letter,head).
link(capital_letter,letter).
link(capital_letter,listexpr).
link(capital_letter,non_terminal).
link(capital_letter,non_unit_clause).
link(capital_letter,quoted_item).
link(capital_letter,quoted_items).
link(capital_letter,sentence).
link(capital_letter,sentences).
link(capital_letter,string_item).
link(capital_letter,string_items).
link(capital_letter,subterm).
link(capital_letter,term).
link(capital_letter,term_read_in).
link(capital_letter,token).
link(capital_letter,unit_clause).
link(capital_letter,variable).
link(c_char,c_char).
link(c_chars,c_chars).
link(c_char,str_literal_1).
link(c_constant,c_constant).
link(c_constant,expression).
link(c_constant,expression_st).
link(c_constant,matched_st).
link(c_constant,primary_expr).
link(c_constant,selection_st).
link(c_constant,statement).
link(c_constant,stmts).
link(c_constant,unmatched_st).
link(c_digit,c_constant).
link(c_digit,c_digit).
link(c_digit,dgts).
link(c_digit,expression).
link(c_digit,expression_st).
link(c_digit,float_const).
link(c_digit,float_const_1).
link(c_digit,fract_const).
```

```
link(c_digit,matched_st).
link(c_digit,primary_expr).
link(c_digit,selection_st).
link(c_digit,statement).
link(c_digit,stmts).
link(c_digit,unmatched_st).
link(char,char).
link(char,chars).
link(char_const,char_const).
link(char,quoted_item).
link(char,quoted_items).
link(chars,chars).
link(char,string_item).
link(char,string_items).
link(cip_program,cip_program).
link(claus,cip_program).
link(claus,claus).
link(claus,sentence).
link(claus,sentences).
link(c_layout_char,c_layout_char).
link(c_layout_char,c_layout_chars).
link(c_layout_char,c_space).
link(c_layout_char,c_spc).
link(c_layout_chars,c_layout_chars).
link(c_layout_chars,c_space).
link(c_layout_chars,c_spc).
link(c_letter,c_letter).
link(c_letter,declaration_2).
link(c_letter,declarator).
link(c_letter,expression).
link(c_letter,expression_st).
link(c_letter,function_def).
link(c_letter,identifier).
link(c_letter,init_decl_list).
link(c_letter,matched_st).
link(c_letter,primary_expr).
link(c_letter,selection_st).
link(c_letter,statement).
link(c_letter,stmts).
link(c_letter,unmatched_st).
link(command,cip_program).
link(command,command).
link(command,directive).
link(command,sentence).
link(command,sentences).
link(comment,comment).
link(comment,token).
link(compound_st,compound_st).
link(compound_st,embedded_C).
link(compound_st,goal).
link(compound_st,goals).
link(compound_st,matched_st).
link(compound_st,selection_st).
link(compound_st,statement).
link(compound_st,stmts).
link(compound_st,unmatched_st).
link(constant,arguments).
link(constant,cip_program).
link(constant,claus).
link(constant,constant).
```

```
link(constant,goal).
link(constant,goals).
link(constant,grammar_rule).
link(constant,gr_body).
link(constant,gr_head).
link(constant,head).
link(constant,listexpr).
link(constant,non_terminal).
link(constant,non_unit_clause).
link(constant,sentence).
link(constant,sentences).
link(constant,subterm).
link(constant,term).
link(constant,term_read_in).
link(constant,unit_clause).
link(c_space,c_space).
link(c_space,c_spc).
link(c_spc,c_spc).
link(declaration_1,declaration).
link(declaration_1,declaration_1).
link(declaration_1,declarations).
link(declaration_2,declaration_2).
link(declaration,declaration).
link(declaration,declarations).
link(declarations,declarations).
link(declarator,declaration_2).
link(declarator,declarator).
link(declarator,function_def).
link(declarator,init_decl_list).
link(decl_spec,decl_spec).
link(decl_spec,decl_specs).
link(decl_spec,decls_specs).
link(decl_spec,function_def).
link(decl_specs,decl_specs).
link(decl_specs,decls_specs).
link(decl_specs,function_def).
link(decls_specs,decls_specs).
link(dgts,c_constant).
link(dgts,dgts).
link(dgts,expression).
link(dgts,expression_st).
link(dgts,float_const).
link(dgts,float_const_1).
link(dgts,fract_const).
link(dgts,matched_st).
link(dgts,primary_expr).
link(dgts,selection_st).
link(dgts,statement).
link(dgts,stmts).
link(dgts,unmatched_st).
link(digit,alpha).
link(digit,alphanumeric).
link(digit,alphanumerics).
link(digit,alphas).
link(digit,arguments).
link(digit,base).
link(digit,char).
link(digit,chars).
link(digit,cip_program).
link(digit,claus).
```

```
link(digit,constant).
link(digit,digit).
link(digit,digits).
link(digit,exponent).
link(digit,float).
link(digit,goal).
link(digit,goals).
link(digit,grammar_rule).
link(digit,gr_body).
link(digit,gr_head).
link(digit,head).
link(digit,integer).
link(digit,listexpr).
link(digit,natural_number).
link(digit,non_terminal).
link(digit,non_unit_clause).
link(digit,number).
link(digit,quoted_item).
link(digit,quoted_items).
link(digits,arguments).
link(digits,base).
link(digits,cip_program).
link(digits,claus).
link(digits,constant).
link(digits,digits).
link(digit,sentence).
link(digit,sentences).
link(digits,exponent).
link(digits,float).
link(digits,goal).
link(digits,goals).
link(digits,grammar_rule).
link(digits,gr_body).
link(digits,gr_head).
link(digits,head).
link(digit,simple_float).
link(digits,integer).
link(digits,listexpr).
link(digits,natural_number).
link(digits,non_terminal).
link(digits,non_unit_clause).
link(digits,number).
link(digits,sentence).
link(digits,sentences).
link(digits,simple_float).
link(digits,subterm).
link(digits,term).
link(digits,term_read_in).
link(digits,token).
link(digit,string_item).
link(digit,string_items).
link(digit,subterm).
link(digits,unit_clause).
link(digits,unsigned_float).
link(digit,term).
link(digit,term_read_in).
link(digit,token).
link(digit,unit_clause).
link(digit,unsigned_float).
link(directive,cip_program).
```

```
link(directive,directive).
link(directive,sentence).
link(directive,sentences).
link(embedded_C,embedded_C).
link(embedded_C,goal).
link(embedded_C,goals).
link(exponent,exponent).
link(exponent_part,exponent_part).
link(expression,expression).
link(expression,expression_st).
link(expression,matched_st).
link(expression,selection_st).
link(expression,statement).
link(expression_st,expression_st).
link(expression_st,matched_st).
link(expression,stmts).
link(expression_st,selection_st).
link(expression_st,statement).
link(expression_st,stmts).
link(expression_st,unmatched_st).
link(expression,unmatched_st).
link(float,arguments).
link(float,cip_program).
link(float,claus).
link(float_const_1,c_constant).
link(float_const_1,expression).
link(float_const_1,expression_st).
link(float_const_1,float_const).
link(float_const_1,float_const_1).
link(float_const_1,matched_st).
link(float_const_1,primary_expr).
link(float_const_1,selection_st).
link(float_const_1,statement).
link(float_const_1,stmts).
link(float_const_1,unmatched_st).
link(float,constant).
link(float_const,c_constant).
link(float_const,expression).
link(float_const,expression_st).
link(float_const,float_const).
link(float_const,matched_st).
link(float_const,primary_expr).
link(float_const,selection_st).
link(float_const,statement).
link(float_const,stmts).
link(float_const,unmatched_st).
link(float,float).
link(float,goal).
link(float,goals).
link(float,grammar_rule).
link(float,gr_body).
link(float,gr_head).
link(float,head).
link(float,listexpr).
link(float,non_terminal).
link(float,non_unit_clause).
link(float,number).
link(float,sentence).
link(float,sentences).
link(float,subterm).
```

```
link(float,term).
link(float,term_read_in).
link(float,unit_clause).
link(fract_const,c_constant).
link(fract_const,expression).
link(fract_const,expression_st).
link(fract_const,float_const).
link(fract_const,float_const_1).
link(fract_const,fract_const).
link(fract_const,matched_st).
link(fract_const,primary_expr).
link(fract_const,selection_st).
link(fract_const,statement).
link(fract_const,stmts).
link(fract_const,unmatched_st).
link(full_stop,full_stop).
link(full_stop,token).
link(funct,arguments).
link(funct,cip_program).
link(funct,claus).
link(funct,funct).
link(funct,goal).
link(funct,goals).
link(funct,grammar_rule).
link(funct,gr_body).
link(funct,gr_head).
link(funct,head).
link(function_def,function_def).
link(funct,listexpr).
link(funct,non_terminal).
link(funct,non_unit_clause).
link(funct,sentence).
link(funct,sentences).
link(funct,subterm).
link(funct,term).
link(funct,term_read_in).
link(funct,unit_clause).
link(goal,goal).
link(goal,goals).
link(goals,goals).
link(grammar_rule,cip_program).
link(grammar_rule,grammar_rule).
link(grammar_rule,sentence).
link(grammar_rule,sentences).
link(gr_body,gr_body).
link(gr_condition,gr_body).
link(gr_condition,gr_condition).
link(gr_head,cip_program).
link(gr_head,grammar_rule).
link(gr_head,gr_head).
link(gr_head,sentence).
link(gr_head,sentences).
link(head,cip_program).
link(head,claus).
link(head,head).
link(head,non_unit_clause).
link(head,sentence).
link(head,sentences).
link(head,unit_clause).
link(identifier,declaration_2).
```

```
% filename : flatten

% flattening a list (ref: [Sterling86]).
%       +   -
flatten(Xs,Ys) :-
   flatten(Xs,[],Ys), !.

flatten([X|Xs],S,Ys) :-
   list(X), flatten(X,[Xs|S],Ys).  flatten([X|Xs],S,[X|Ys])
:-
   (atom(X); number(X)), X \== [], flatten(Xs,S,Ys).
flatten([],[X|S],Ys) :-
   flatten(X,S,Ys).
flatten([],[],[]).

list([_|_]).
list([]).
```

2

VITA

Lukas Budianto Santoso

Candidate for the Degree of

Master of Science

Thesis: DETAILED DESIGN AND PARTIAL IMPLEMENTATION OF A PRE-PROCESSOR FOR PROLOG PROGRAMS WITH EMBEDDED C STATEMENTS

Major Field: Computer Science

Biographical:

Personal Data: Born in Semarang, Indonesia, October 18, 1962, the son of Mr. Slamet Santoso and Mrs. Sartini Santoso.

Education: Graduated from Loyola High School, Semarang, Indonesia, in April 1980; received Sarjana Teknik degree with a major in Electrical Engineering from Bandung Institute of Technology, Bandung, Indonesia in March 1986; completed requirements for the Master of Science degree at Oklahoma State University in July 1991.

Professional Experience: Junior System Analyst and Programmer, PT. Pan Systems, Jakarta, Indonesia, 1986 - 1987.