# TWS: A C++ TRANSLATOR WRITING SYSTEM DESIGNED FOR THE INCREMENTAL APPROACH TO TEACHING TRANSLATION

By

SRIVATSAN MADHAVAN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

This document is dedicated to my *alma mater* – The University of Florida.

ACKNOWLEDGMENTS

I thank my advisor Dr. Manuel Bermudez for inspiring me to do this work. I would also like to thank Kajal Jain for all her support.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS ..... iv

LIST OF TABLES ..... viii

LIST OF FIGURES ..... ix

ABSTRACT ..... x

CHAPTER

1 INTRODUCTION ..... 1

The State of the Compilers Course ..... 1
The Incremental Approach ..... 2

2 LANGUAGES, GRAMMARS, AND COMPILER CONSTRUCTION ..... 4

Formal Languages ..... 4
Grammars ..... 4
Types of Grammars (Chomsky's Classification) [2] ..... 5
Backus-Naur Form (BNF) ..... 6
Syntax: Parsing ..... 7
Semantics and Code Generation ..... 8
Parse Trees and Attribute Grammar ..... 9
Attribute Grammar ..... 10

3 OVERVIEW OF EXISTING TOOLS ..... 12

Lex and Flex ..... 12
Sample Scanner ..... 13
Format of the Input File ..... 13
Patterns ..... 15
Scanner Generated by *Flex* ..... 15
Yacc and Bison ..... 17
Bison Grammar File ..... 17
Grammar Rules Section ..... 17
Recursive Rules ..... 19
Semantics in Bison ..... 20
Actions ..... 20

4 TRANSLATOR WRITING SYSTEM ..... 22

Overall Organization ..... 22
*pgen* – The Parser Generator ..... 24
Abstract Syntax Tree Transforms ..... 25
Regular Expression Operators Supported by the TWS ..... 25
The transformations ..... 26
The `*` Operator ..... 26
The + Operator ..... 26
The `list` Operator ..... 27
The `?` Operator ..... 27
Constrainer and the Code Generator Framework ..... 27

5 SAMPLE COURSE DEVELOPMENT ..... 28

Initial Translator ..... 28
Adding Operators ..... 29
Changes to the Lexical Analyzer – *lex.tiny* ..... 29
Changes to the Parser – *parse.tiny* ..... 29
Modifications to the Constrainer ..... 30
Changes to the Code Generator ..... 32
Adding Statements ..... 32
Changes to the Lexical Analyzer – *lex.tiny* ..... 32
Changes to the Parser – *parse.tiny* ..... 33
Modifications to the Constrainer ..... 33
Changes to the Code Generator ..... 33
Conclusions and Future Work ..... 34

APPENDIX

A THE PGEN META GRAMMAR ..... 35

B SOURCE CODE FOR *lex.tiny* and *parse.tiny* ..... 41

lex.tiny ..... 41
parse.tiny ..... 43

C SOURCE CODE FOR THE INITIAL CONSTRAINER AND CODE GENERATOR ..... 44

nodes.h ..... 44
tws::constrainer class ..... 45
constrainer.h ..... 45
constrainer.cpp ..... 45
tws::codegenerator class ..... 51
codegen.h ..... 51
codegen.cpp ..... 51

LIST OF REFERENCES ..... 55

BIOGRAPHICAL SKETCH ..... 56

LIST OF TABLES

Table page

3-1. Regular expression patterns and corresponding matched expressions.....16

4-1. Regular expression operators and their meanings as supported by the TWS.....25

LIST OF FIGURES

Figure page

2-1. Parse tree....9

2-2: Grammar for arithmetic expressions....10

2-3: Attribute grammar for constant expressions, using the standard arithmetic operations....11

3-1. Sample *flex* input....13

3-2: Flex input sections....13

3-3 Format of bison grammar file....17

5-1 – *parse.tiny*: The parser specification of the initial *Tiny* language....29

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

TWS: A C++ TRANSLATOR WRITING SYSTEM DESIGNED FOR THE INCREMENTAL APPROACH TO TEACHING TRANSLATION

By

Srivatsan Madhavan

August 2003

Chair: Dr. Manuel E. Bermudez
Major Department: Computer and Information Science and Engineering

TWS is a C++ framework for teaching compiler construction that focuses on the underlying principles of translation, syntax recognition, and semantic processing. This system facilitates a teaching approach that resembles a spiral, in which every component of the compiler is visited to add new features or constructs to the language. This is in contrast with the traditional approach with a system in which each of the lexical analyzer, parser, semantic analyzer, and code generator is fully specified at the beginning and the learning process concentrates on each of these tasks one at a time.

# CHAPTER 1
# INTRODUCTION

This chapter briefly introduces the incremental approach to teaching compiler construction [1] and the relevance of this thesis to such an approach.

## The State of the Compilers Course

A little more than a decade ago, in computer science curricula at major universities around the world, a course in compiler construction was considered indispensable to the formation of the undergraduate student. It was the rare computer science academic program that did not have a required compiler construction course in its curriculum. This situation has now changed. For the last decade or so, the compilers course in most computer science curricula has been in decline. There are many reasons for this.

First, there is the maturity of the compiler discipline itself. Compiler construction techniques have evolved at a much slower pace in recent years, and computer science curricula tend to give higher importance to emerging technologies.

Another reason for the decline is the proliferation of languages and paradigms in recent years, prompting CS departments, curriculum experts, and textbook writers to focus their efforts on issues of design and implementation of programming languages, rather than the implementation techniques traditionally covered in a compilers course. As a result, CS curricula today are much more likely to require a programming languages course, than a compilers course.

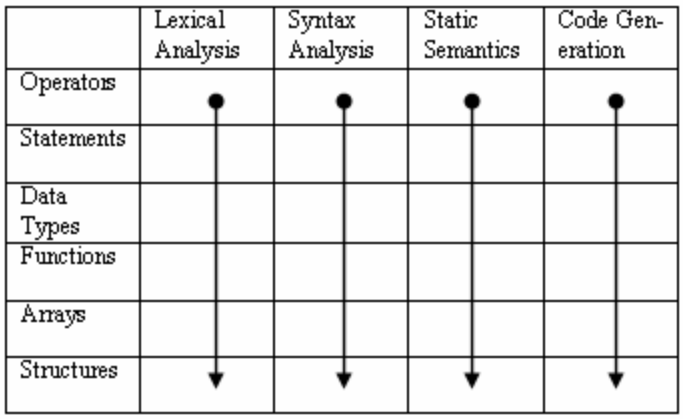


Figure 1-1. Traditional Compiler course design

The principal problem with compiler courses in the past has been that the focus is on compilation which as a skill is no longer very fundamental. The underlying principles of translation, including syntax recognition and semantic processing, *transcend compilation.* In addition, the decreasing popularity of the compilers course is due to the mismatch between the sequence of course topics and the sequence of implementation efforts. The structure of a typical approach to teaching translation is shown in Figure 1-1.

**The Incremental Approach**

In the new design proposed for teaching the compilers course[1], the emphasis is on *translation*. A good number of compiler specific issues are removed from the course. The course design revolves around a project which consists of maintaining and extending an *initial* compiler, rather than implementing one from scratch. The initial compiler will have a highly extensible and modifiable design and could be implemented in a language like C++, Java, or C#.

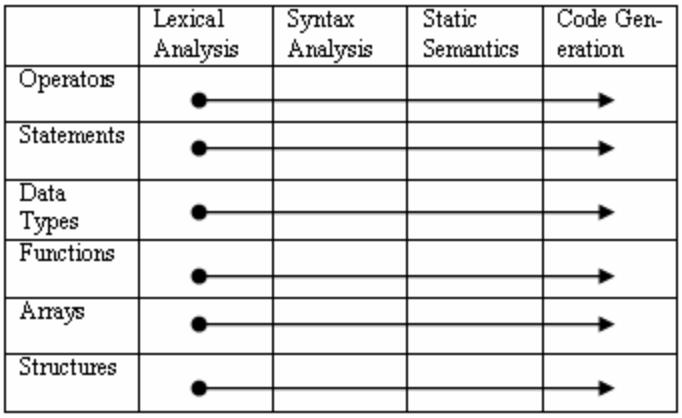


Figure 1-2. New Compiler course design

The new approach resembles a spiral: students repeatedly visit every component of the compiler (scanner, parser, contextual constrainer, code generator) to add new constructs or features. With each visit, the student gains deeper understanding of the translator's architecture, components, and structure. Thus, the progress is from the simple concepts to the complex, rather than from "front" to "back" of the compiler.

This thesis presents the design and implementation of one particular initial compiler, written in C++, which can be used to support such a course.

In Chapter 2, we introduce the mathematical concepts underlying compiler and translator writing. A brief overview of languages, grammars, and compiler construction techniques is presented here.

Chapter 3 presents a brief tour of tools used in compiler construction. *Bison* and *Flex* are explained in some detail.

In Chapter 4, the translator writing system is introduced and overview of its architecture is presented.

# CHAPTER 2
# LANGUAGES, GRAMMARS, AND COMPILER CONSTRUCTION

This chapter gives a brief introduction to preliminary concepts of compiler construction, different types of languages, the most common parsing techniques and techniques for representing semantics and code generation.

## Formal Languages

Consider algebraic expressions written with the symbols A = {x, y, z, +, *, (, ) }. The following are some of them: “x + y”, “z * (y + z)”, “x”. The following are, however, not legitimate algebraic expressions as they contain *syntax error’s*: “(x”, “x + * y”, “y*+x”. Syntactically correct algebraic expressions constitute a subset of the whole set $A^*$ of possible strings over A.

In general, given a finite set A (the *alphabet*), a (*formal*) language over A is a subset of $A^*$ (set of strings of A).

## Grammars

A way to specify the structure of a language is with a *grammar*. To define a grammar, we need two kinds of symbols: *non-terminals* and *terminals. Non-terminal* symbols are used to represent a given subset of the language, and *terminal* symbols are final symbols that appear in the strings of the language.

For instance, in the above example, the *terminals* are the symbols appearing in the set A = { x, y, z, +, *, (, ) }. The *non-terminal* symbols can be chosen to represent complete algebraic expressions (E) or terms (T) consisting of factors (F). Then we can say that the algebraic expression E consists of a single term (T)

E → T

or the sum of an algebraic expression and a term

E → E + T

A term may consist of a factor or product of a term and a factor

T → F
T → F * T

A factor may consist of an algebraic expression within parenthesis or an isolated terminal symbol.

F → ( E )
F → x
F → y
F → z

These rules are called *productions* and define the set of all legal strings belonging to this language, and they give us a means to systematically generate such legal strings.

In general, a *phrase-structure grammar* (or simply grammar) G consists of

1. A finite set N of *nonterminal symbols,*
2. A finite set T of *terminal symbols*, where $N \bigcap T = \phi$,
3. A finite subset P of $[(N \cup T)^* - T^*] \times (N \cup T)^*$ called the set of productions, and
4. A starting symbol $\sigma \in N$.

We write $G = (N, T, P, \sigma)$

**Types of Grammars (Chomsky's Classification) [2]**

Let G be a grammar. Let $\lambda$ denote the null string.

0. G is a *phrase-structure* (or type-0) grammar if every production is of the form:

$$\alpha \to \beta,$$

where $\alpha \in (N \cup T)^* - T, \beta \in (N \cup T)^*$.

1. G is a *context-sensitive* (or type-1) grammar if every production is of the form:

$$\alpha A\beta \to \alpha\delta\beta,$$

where $\alpha, \beta \in (N \cup T)^*, A \in N, \delta \in (N \cup T)^* - \{\lambda\}$.

2. G is a *context-free* (or type-2) grammar if every production is of the form:

$$A \to \delta,$$

where $A \in N, \delta \in (N \cup T)^*$.

3. G is a *regular* (or type-3) grammar if every production is of the form:

$$A \to a \text{ or } A \to aB \text{ or } A \to \lambda,$$

where $A, B \in N, a \in T$.

A language L is *context-sensitive* (respectively *context-free, regular*) if there is a context-sensitive (respectively context-free, regular) grammar G such that L = L(G).

**Backus-Naur Form (BNF)**

The Backus-Naur form [3] is a notation for writing productions. The production S→ T is written as S ::= T. Productions of the form S ::= $T_1$, S ::= $T_2$, … , S ::= $T_n$ can be combined as

$$S ::= T_1|T_2|\ldots|T_n.$$

The Extended Backus-Naur Form (EBNF) is any variation of the basic BNF metasyntax notation with (some of) the following additional constructs:

- Square brackets "[ … ]" surrounding optional items

- Suffix "*" for Kleene closure (a sequence of zero or more items), suffix "+" for one or more of an item, curly braces enclosing a list of alternatives.
- Super/subscripts indicating between *n* and *m* occurrences. e.g. $r_5^2$ denotes anywhere between 2 and 5 *r*'s.

**Syntax: Parsing**

A parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a *symbol*. Those which are built by grouping smaller constructs according to grammatical rules are called *nonterminal symbols*; those which can't be subdivided are called *terminal symbols* or *token types*. We call a piece of input corresponding to a single terminal symbol a *token*, and a piece corresponding to a single nonterminal symbol a *grouping*.

There are three types of parsers for grammars.

1. Universal parsing methods such as the Cocke-Younger-Kasami algorithm[4],
2. Top-down parsers [2], and
3. Bottom-Up parsers [2].

The most efficient top-down and bottom-up methods work only on sub-classes of grammars, but several of these subclasses, such as the LL and LR grammars [2] are expressive enough to describe most syntactic constructs in programming languages. Parsers implemented by hand (most typically recursive descent parsing) often work with

LL grammars. Parsers for the larger class of LR grammars are usually constructed by automated tools.

The most common formal system for presenting the grammar for the source language for humans to read is *Backus-Naur Form* or *BNF*, which was described in the previous section. Any grammar expressed in BNF is a context-free grammar or is equivalent to one.

There are various important subclasses of context-free grammar. SLR(1) grammars are those in which it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Most LR(1) grammars are also LALR(1) grammars.

Parsers for LALR(1) grammars are *deterministic*, meaning roughly that the next grammar rule to apply at any point in the input is uniquely determined by the preceding input and a fixed, finite portion (called a *look-ahead*) of the remaining input. A context-free grammar can be *ambiguous*, meaning that there are multiple ways to apply the grammar rules to get the some inputs. Even unambiguous grammars can be *non-deterministic*, meaning that no fixed look-ahead always suffices to determine the next grammar rule to apply.

## Semantics and Code Generation

Syntax concerns the *form* of the valid program, while semantics concerns its *meaning*. It is conventional to say that the syntax of a language is precisely that portion of the language definition that can be described conveniently by a context-free grammar, while the semantics is that portion of the definition that cannot. Both semantic analysis

and (intermediate) code generation can be described in terms of annotation or *decoration* of a parse tree or syntax tree. The notations themselves are known as *attributes.*

**Parse Trees and Attribute Grammar**

Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents. The ways in which these constituents combine are defined by a set of potentially recursive rules, which is the context free grammar.

Consider the grammar of algebraic expressions presented earlier. A valid string from this language would be "x * y + z * (x + y)". The corresponding parse tree appears in Figure 2-1.

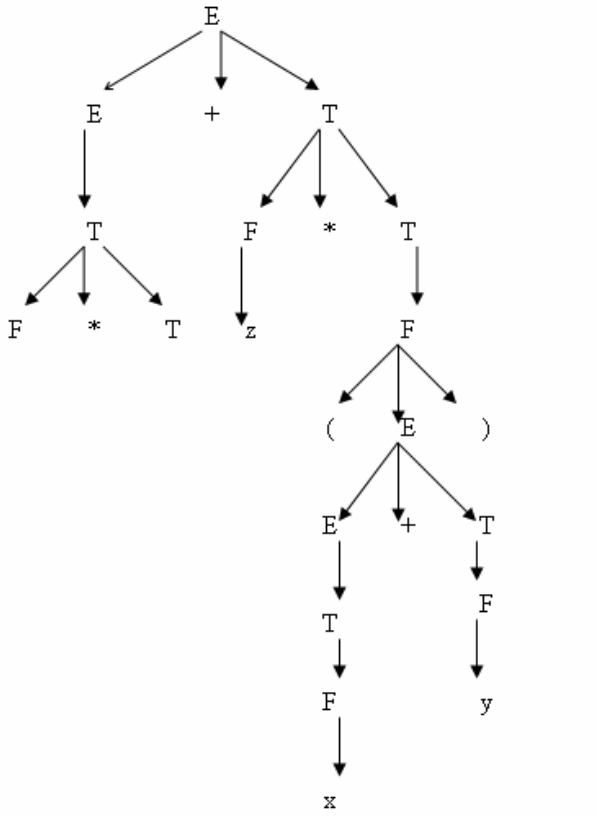


Figure 2-1. Parse tree

**Attribute Grammar**

When programming languages (or arithmetic expressions) are expressed as a CFG, the grammar fails to say anything about the meaning of the program. An attribute grammar can be used to specify this meaning. It is common to associate an attribute with each symbol (both terminal and non-terminal) which is called, say, *val* and the grammar is augmented with a set of rules for each production, to specify how the *vals* of different symbols are related. The resulting grammar is called an *attribute grammar*.

Consider the grammar for arithmetic expressions composed of constants, with precedence and associativity, shown in Figure 2-2. A simple attribute grammar for this grammar is given in Figure 2-3.

E → E + T
E → E – T
E → T
T → T * F
T → T / F
T → F
F → - F
F → ( E )
F → *const*

Figure 2-2: Grammar for arithmetic expressions

1. $E_1 \rightarrow E_2 + T$
   ▷ $E_1$.val := sum($E_2$.val , T.val)
2. $E_1 \rightarrow E_2 - T$
   ▷ $E_1$.val := difference($E_2$.val , T.val)
3. $E \rightarrow T$
   ▷ E.val := T.val
4. $T_1 \rightarrow T_2 * F$
   ▷ $T_1$.val := product($T_2$.val, F.val)
5. $T_1 \rightarrow T_2 / F$
   ▷ $T_1$.val := quotient($T_2$.val, F)
6. $T \rightarrow F$
   ▷ T.val := F.val
7. $F_1 \rightarrow - F_2$
   ▷ $F_1$.val := additive_inverse($F_2$.val)
8. $F \rightarrow ( E )$
   ▷ F.val := E.val
9. $F \rightarrow$ *const*
   ▷ F.val := *const*.val

Figure 2-3: Attribute grammar for constant expressions, using the standard arithmetic operations

# CHAPTER 3
# OVERVIEW OF EXISTING TOOLS

In this chapter, we present an overview of popular compiler construction tools available for C and C++. *Lex* and *Yacc*, and their popular reimplementations, *Flex*[5] and *Bison* [6] are the de-facto standard among compiler construction tools.

## Lex and Flex

*Flex,* Fast lexical analyzer generator is a GNU tool used to produce programs that perform pattern-matching on text. It takes as input a description of the scanner it is going to generate. The description is in the form of pairs of regular expressions and C code. *Flex* has support for C++, so the description can also be given as C++ code. The output generated by *Flex* is in the form of C (or C++) source code which defines a function called *yylex()*. Whenever a valid token is found, *yylex()* executes the corresponding C or C++ code.

**Sample Scanner**

The example in Figure 3.1 generates a program which will count the number of lines and the number of characters in the standard input and report it on the console.

```
int num_lines = 0, num_chars = 0;

%%
\n        ++num_chars;++num_lines;
.         ++num_chars;

%%
int main()
{
     yylex();
     printf("# of lines:%d\n",num_lines);
     printf("$ of chars:%d\n",num_chars);
     return 0;
}
```

Figure 3-1. Sample *flex* input

**Format of the Input File**

The *flex* input file consists of three sections, separated by a line consisting only of `%%` , as shown in Figure 3-2.

```
definitions
%%
rules
%%
user code
```

Figure 3-2: Flex input sections

The *definitions* section contains declarations of simple *name* definitions to simplify the scanner specification and declarations of *start conditions*.

```
name definition
```

The *name* is a word beginning with a letter or an underscore (“_”) followed by zero or more letters, digits, “_”, or “-”(dash). The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using *{name}*, which will expand to *(definition)*. For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines DIGIT to be a regular expression which matches a single digit, and ID to be a regular expression which matches a letter followed by zero or more letters or digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one or more digits followed by a “.” followed by zero or more digits.

The *rules* section of the *flex* input contains a series of rules of the form:

```
pattern   action
```

where the pattern must be unindented and the action must begin on the same line.

Finally, the user code section is simply copied to `lex.yy.c` verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%` in the input file may be skipped as well.

In the definitions and rules sections, any indented text or text enclosed in `%{` and `%}` is copied verbatim to the output (with the `%{}`'s removed). The `%{}`'s must appear unindented on lines by themselves.

In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables. These are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered.

In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with `/*`) is also copied verbatim to the output, up to the next `*/`.

**Patterns**

Patterns in the input are written using an extended set of regular expressions. Some examples are shown in Table 3-1

**Scanner Generated by *Flex***

The output of *flex* is the file `lex.yy.c`, which contains the scanning routine `yylex()`, a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, `yylex()` is declared as follows:

```
int yylex()
{
    ... various definitions and the actions in here ...
}
```

Whenever `yylex()` is called, it scans tokens from the global input file `yyin` (which defaults to `stdin`). It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions executes a `return` statement. If `yylex()` stops scanning due to executing a `return` statement in one of the actions, the scanner may later be called again and it will resume scanning where it left off.

Table 3-1. Regular expression patterns and corresponding matched expressions

| Pattern | Matched Expression |
|---|---|
| X | Match the character ‘x’ |
| . | Any character Except newline |
| [xyz] | A character class, in this case, either an ‘x’ or a ‘y’ or a ‘z’. |
| [abj-oz] | A character class with a range in it, in this case, an ‘a’, or ‘b’, or any letter from ‘j’ to ‘o’ or a ‘z’ |
| [^A-Z] | A negated character class, i.e., any character not in the character class ‘[A-Z]’ |
| r* | Zero or more *r*’s, where *r* is any valid regular expression |
| r+ | One or more *r*’s |
| r? | Zero or one *r*’s (i.e., an optional *r*) |
| r{2,5} | Anywhere from two to five *r*’s |
| r{2,} | Two or more *r*’s |
| r{4} | Exactly four r’s |
| {name} | The expansion of the *name*definition |
| ”[xyz]\”foo” | The literal string [xyz]”foo |
| \x | If x is an ‘a’, ‘b’, ‘f’, ‘n’, ‘r’, ‘t’ or ‘v’, then the ANSI C interpretation of \x. Otherwise a literal ‘x’ (used to escape operators such as ‘*’ ) |
| \0 | A NUL character |
| \123 | A character with octal value 123 |
| \x2a | The character with hexadecimal value 2a |
| (r) | Match an ‘r’, parenthesis are used to override precedence |
| Rs | The regular expression ‘r’ followed by the regular expression ‘s’ (“concatenation”) |
| r\|s | Either an ‘r’ or an ‘s’ |
| r/s | An r but only if it is followed by an s |
| ^r | An r, but only at the beginning of a line |
| r$ | An r, but only at the end of a line |
| <s>r | An r, but only in start condition s. |
| <*>r | An r in any start condition, even an exclusive one |
| <<EOF>> | And end-of-file |
| <s1,s2><<EOF>> | An end of file when in start condition s1 or s2 |

## Yacc and Bison

*Yacc,* Yet Another Compiler Compiler [7], is a tool written in portable C which accepts a language specified as LALR(1) grammar with disambiguating rules and generates a parser for this language. *Bison* is a parser generator that is completely compatible with *Yacc.* It is capable of generating C as well as C++ code for the parser.

### Bison Grammar File

Bison takes as input a context-free grammar specification and produces a C or a C++ language function that recognizes correct sentences generated in the language generated by the grammar.

The Bison grammar input file conventionally has a name ending in `.y or .ypp`, as shown in Figure 3-3.

```
%{
C/C++ declarations
%}

Bison declarations

%%
Grammar rules
%%

Additional C/C++
code
```

Figure 3-3 Format of bison grammar file

### Grammar Rules Section

The *grammar rules* section contains one or more Bison grammar rules and nothing else.

There must always be at least one grammar rule, and the first `%%` (which precedes the grammar rules) may never be omitted even if it is the first item in the file.

Bison grammar rules have the following general form:

```
result: components...
          ;
```

where *result* is the nonterminal symbol that this rule describes and *components* are various terminal and nonterminal symbols that are put together by this rule.

For example,

```
exp:     exp '+' exp
       ;
```

says that two groupings of type `exp`, with a + token in between, can be combined into a larger grouping of type `exp`.

Whitespace in rules is significant only to separate symbols.

Scattered among the components can be *actions* that determine the semantics of the rule. An action looks like this:

{*C/C++ statements*}

Usually there is only one action and it follows the components.

Multiple rules for the same *result* can be written separately or can be joined with the vertical-bar character `|` as follows:

```
result:    rule1-components...
         | rule2-components...
         ...
         ;
```

They are still considered distinct rules even when joined in this way. If *components* in a rule is empty, it means that *result* can match the empty string. For example, here is how to define a comma-separated sequence of zero or more `exp` groupings:

```
expseq:   /* empty */
        | expseq1
        ;

expseq1:  exp
        | expseq1 ',' exp;
```

**Recursive Rules**

A rule is called *recursive* when its *result* nonterminal appears also on its right hand side. Nearly all Bison grammars need to use recursion, because that is the only way to define a sequence of any number of somethings. Consider the following recursive definition of a comma-separated sequence of one or more expressions:

```
expseq1:  exp
        | expseq1 ',' exp
        ;
```

Since the recursive use of `expseq1` is the leftmost symbol in the right hand side, we call this *left recursion*. By contrast, here the same construct is defined using *right recursion*:

```
expseq1:  exp
        | exp ',' expseq1
        ;
```

Any kind of sequence can be defined using either left recursion or right recursion, but one should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space. Right recursion uses up space on the Bison stack in proportion to the number of elements in the sequence, because all the elements must be shifted onto the stack before the rule can be applied even once.

*Indirect* or *mutual* recursion occurs when the result of the rule does not appear directly on its right hand side, but does appear in rules for other nonterminals which do appear on its right hand side.

For example:

```
expr:     primary
        | primary '+' primary
        ;

primary:  constant
        | '(' expr ')'
        ;
```

defines two mutually-recursive nonterminals, since each refers to the other.

**Semantics in Bison**

The grammar rules for a language determine only the syntax. The semantics are determined by the semantic values associated with various tokens and groupings, and by the actions taken when various groupings are recognized. These values are similar to attributes presented in the context of *attribute grammars.*

In a simple program it may be sufficient to use the same data type for the semantic values of all language constructs.

Bison's default is to use type `int` for all semantic values. To specify some other type, `YYSTYPE` macro should be redefined. For example, to redefine the default value of the semantic values to *double,* we use

```
#define YYSTYPE double
```

**Actions**

An action accompanies a syntactic rule and contains C/C++ code to be executed each time an instance of that rule is recognized. The task of most actions is to compute a semantic value for the grouping built by the rule from the semantic values associated with tokens or smaller groupings.

An action consists of statements surrounded by braces, much like a compound statement in C++. It can be placed at any position in the rule; it is executed at that position. Most rules have just one action at the end of the rule, following all the components. Actions in the middle of a rule are tricky and used only for special purposes.

The C++ code in an action can refer to the semantic values of the components matched by the rule with the construct `$n`, which stands for the value of the *n*th component. The semantic value for the grouping being constructed is `$$`. (Bison translates both of these constructs into array element references when it copies the actions into the parser file.)

Here is a typical example:

```
exp:     ...
         | exp '+' exp
             { $$ = $1 + $3; }
```

This rule constructs an `exp` from two smaller `exp` groupings connected by a plus-sign token. In the action, `$1` and `$3` refer to the semantic values of the two component `exp` groupings, which are the first and third symbols on the right hand side of the rule. The sum is stored into `$$` so that it becomes the semantic value of the addition-expression just recognized by the rule. If there were a useful semantic value associated with the `` `+' `` token, it could be referred to as `$2`.

If an action is not specified for a rule, *Bison* supplies a default: `$$ = $1`. Thus, the value of the first symbol in the rule becomes the value of the whole rule. The default rule is valid only if the two data types match. There is no meaningful default action for an empty rule; every empty rule must have an explicit action unless the rule's value does not matter.

# CHAPTER 4
# TRANSLATOR WRITING SYSTEM

This chapter discusses the organization of the Translator Writing System, *pgen* – the parser generator, the abstract syntax tree transformations used to convert regular grammar to valid *bison* input and the input files required to use this system.

## Overall Organization

The heart of the scheme shown in Figure 4-1 is the *pgen* parser generator. This parser generator takes an input grammar specification and converts it into an input format that can be consumed by *bison* to produce a functional parser. *pgen* is a full fledged compiler in itself. It takes as input the grammar of the language as augmented EBNF (described in the following sections) and parses it and applies transformations to the input grammar to convert it to a syntax that is compatible with *bison*. A major step in this conversion involves converting the productions specified as regular expressions to non-regular expression forms – i.e., removal of `+`, `*`,`?`, and `list` operators from the input.

The lexical analyzer is generated using conventional *flex* and is directly integrated into the parser which is finally generated by *bison.*

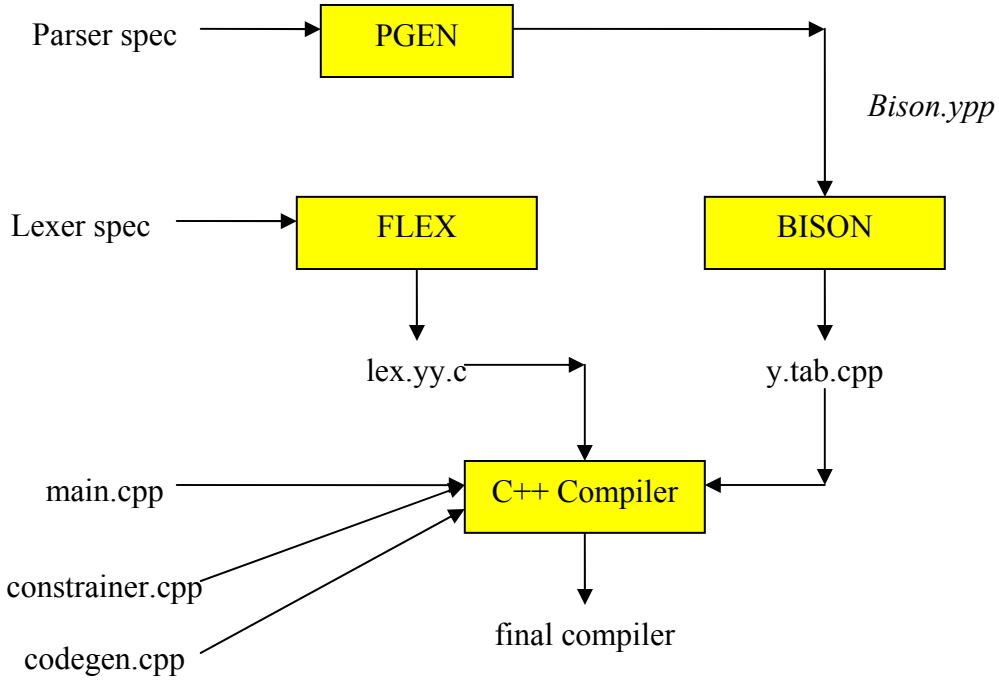


Figure 4.1 – Organization of TWS

Other supporting modules and the main framework specified by *main.cpp* are compiled with the lexer and the parser to finally produce the compiler. *main.cpp* has skeletal code that can be modified to specify the semantics of the language specified in by the grammar. The support modules are mainly comprised of classes that support tree data structures to represent the Abstract Syntax Tree (AST), functions providing further operations of the tree data structure like decorating the tree and walking the tree to generate code, data structures supporting the symbol table and the code generator.

### *pgen* – The Parser Generator

The organization of *pgen* is illustrated in Figure 4-2. *pgen* is constructed using conventional compiler construction tools – *bison* and *flex*. It might be possible to compile (or construct) *pgen* using *pgen* itself, but that has not been attempted. The two very important components in generating *pgen* are the *bison* input specification files – Parser.ypp and *pgen.cpp. Parser.ypp* is presented in the Appendix A. *pgen.cpp* provides

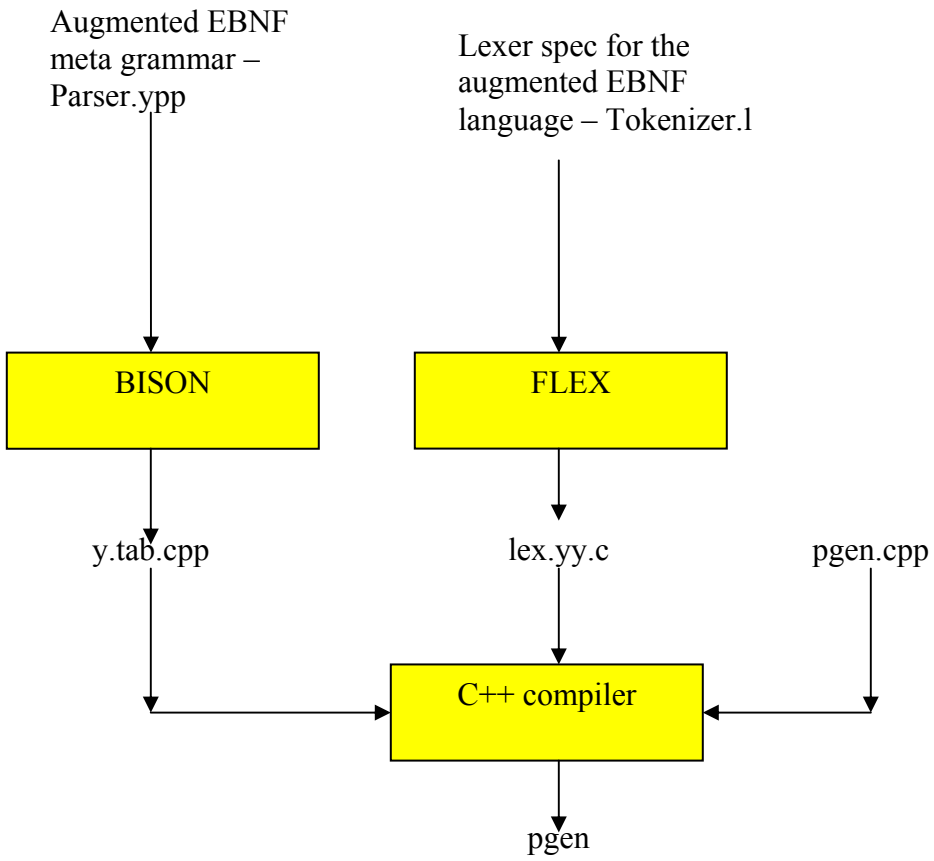


Figure 4-2 – PGEN, The parser Generator

the definitions for the semantic actions specified in *Parser.ypp*. *pgen.cpp* uses the same tree classes, symbol tables, and tree walking routines used by *main.cpp* in the TWS.

## Abstract Syntax Tree Transforms

The major part of work done by *pgen* is in converting the AST of the input grammar to a form that does not use regular expressions. This section describes the AST transformations used by *pgen*.

### Regular Expression Operators Supported by the TWS

Four regular expression operators are supported by TWS. These are shown in Table 4-1. Apart from these, it is also possible to group items together using parentheses.

Table 4-1. Regular expression operators and their meanings as supported by the TWS

| Operator | Meaning | Illustration |
|---|---|---|
| * | Kleene Star – Zero or more instances of | A → B* |
| + | One or more instances of | A → $B^+$ |
| ? | Zero or one instance of - i.e., optional | A → B? |
| list | list operator | A → B list ‘,’<br>List of B’s separated by ‘,’ |

The sample grammar in Figure 4-3 illustrates the usage of these operators.

```
%%
Dclns -> VAR (Dcln ';')*          => "dclns"
           ->                           => "dclns";


Dcln       -> Name list ',' ':' Type=> "dcln";
```

Figure 4-3: Sample TWS grammar specification

In this example, `Dclns` uses the * operator. The entity `(Dcln ';')` is optional, and may appear any number of times. All of the `Dcln` 's will be parsed into one AST node `dclns`.

`Dcln` itself is comprised of a list of `Name`'s separated by `,`(comma) finally terminated by a `:` and a `Type.` Each `Dcln` generates a `dcln` tree node.

**The transformations**

**The * Operator**

```
A -> B*
```

is transformed into

```
A  -> A'
A' -> B A
A' -> ε
```

Where $\varepsilon$ is the null string.

**The + Operator**

```
A  -> B+
```

is transformed into

```
A  -> A'
A' -> B A
A' -> B
```

**The `list` Operator**

```
A  -> B list C
```

is transformed into

```
A  -> B (C B)*
```

which is further transformed using the rules for '*'.

**The `?` Operator**

```
A  -> B?
```

is transformed into

```
A  -> A'
A' -> B
A' -> ε
```

After these transformations have been applied, it is possible to generate a valid *bison* input specification. The program generated by *bison* will parse the input program and generate a parse tree. This parse tree will be walked by the supporting routines to generate code.

## Constrainer and the Code Generator Framework

The *pgen* program generates a *bison* source which in turn generates a parser. This parser emits a file that contains the parse tree (the abstract syntax tree) for the source program. This parse tree is further processed by a constrainer (semantic analyzer) and then by a code generator. These components are specific to the input language and the TWS provides an *initial* constrainer and code generator for *Tiny*, a procedural language similar to *Pascal*. These modules are further discussed in the next chapter.

.

# CHAPTER 5
# SAMPLE COURSE DEVELOPMENT

As was previously mentioned, the new approach to teaching compilers resembles a spiral: every component of the compiler (scanner, parser, contextual constrainer, code generator) is repeatedly visited, to add new constructs or features. This chapter presents a sample course development in which we demonstrate that the TWS system achieves this goal of incremental development by iterating over each of the above mentioned steps.

Excerpts are shown from *parse.tiny*, the grammar specification of the *Tiny* language, *tws::constrainer* and *tws::codegenerator* classes. The sources listings are included in the Appendix C.

## Initial Translator

The grammar for the initial translator, *tiny.parse,* is shown in Figure 5-1. *Tiny* has two data types – *Integer* and *Boolean.* It also supports compound statements enclosed within a BEGIN-END block, variable declaration, assignment, IF-THEN-ELSE conditionals, WHILE loop, and basic integer arithmetic support for addition, subtraction, and negation (unary minus). It also allows reading and writing integers to the console.

The lexical analyzer (flex input), constrainer and code generator are shown in the appendices. In the next two sections, we see how additions can be made to the language and what changes required to the constrainer and the code generator to build a compiler for the new language.

```
%%
Tiny        -> PROGRAM Name ':' Dclns Body Name '.'          => "program";
Dclns       -> VAR (Dcln ';')*                               => "dclns"
            ->                                               => "dclns";
Dcln        -> Name list ',' ':' Type                        => "dcln";
Type        -> INTEGER                                       => "integer"
            -> BOOLEAN                                       => "boolean";
Body        -> BEGINX Statement list ';' END                 => "block";
Statement   -> Name ASSIGNMENT Expression                    => "assign"
            -> OUTPUT '(' Expression ')'                     => "output"
            -> IF Expression THEN Statement
                  ELSE Statement                                    => "if"
            -> WHILE Expression DO Statement                 => "while"
            -> Body
             ->                                              => "<null>";

Expression -> Term
             -> Term LTE Term                               => "<=";

Term        -> Primary
             -> Primary '+' Term                            => "+";

Primary     -> '-' Primary                                  => "-"
             -> READ                                        => "read"
            -> Name
            -> INTEGER_NUM                                  => "<integer>"
            -> '(' Expression ')';

Name        -> IDENTIFIER                                   => "<identifier>";
```

Figure 5-1 – *parse.tiny*: The parser specification of the initial *Tiny* language

## Adding Operators

The following changes are made to add a new operator for multiplication – * to the language.

### Changes to the Lexical Analyzer – *lex.tiny*

The entries for operators in the rules section of lex.tiny are as follows:

```
"+"             { return rule(yytext[0]); }
"-"             { return rule(yytext[0]); }
```

To add the '*' operator, the following line is added:

```
"*"             { return rule(yytext[0]); }
```

### Changes to the Parser – *parse.tiny*

The part of the parser that deals with arithmetic expressions is excerpted here:

```
Expression -> Term
           -> Term LTE Term                    => "<=";

Term       -> Primary
           -> Primary '+' Term                 => "+";

Primary    -> '-' Primary                      => "-"
           -> READ                             => "read"
           -> Name
           -> INTEGER_NUM                      => "<integer>"
           -> '(' Expression ')';

Name       -> IDENTIFIER                       => "<identifier>";
```

To introduce a multiplicative operator, a new production Factor is introduced:

```
.
Expression -> Term
           -> Term LTE Term                    => "<=";

Term       -> Factor
           -> Factor '+' Term                  => "+";

Factor     -> Primary
           -> Primary '*' Factor               => "*";

Primary    -> '-' Primary                      => "-"
           -> READ                             => "read"
           -> Name
           -> INTEGER_NUM                      => "<integer>"
           -> '(' Expression ')';

Name       -> IDENTIFIER                       => "<identifier>";
```

**Modifications to the Constrainer**

The constrainer consists of a class tws::constrainer and a header nodes.h which defines the tree nodes on which the constrainer and the code generator have to act. nodes.h has the following entries:

```
addnode(ProgramNode,"program");
addnode(TypesNode, "types");
addnode(TypeNode, "type");
addnode(DclnsNode ,"dclns");
addnode(DclnNode, "dcln");
addnode(IntegerTNode,"integer");
addnode(BooleanTNode, "boolean");
addnode(BlockNode,"block");
addnode(AssignNode, "assign");
addnode(OutputNode, "output");
addnode(IfNode ,"if");
```

```
addnode(WhileNode ,"while");
addnode(NullNode , "null");
addnode(LENode ,"<=");
addnode(PlusNode ,"+");
addnode(MinusNode,"-");
addnode(ReadNode,"read");
addnode(IntegerNode,"<integer>");
addnode(IdentifierNode,"<identifier>");
```

A new entry for the * node will be added as follows:

```
addnode(MultNode ,"*");
```

*tws::constrainer* has a method *expression()* which analyzes expressions in the AST and determines if it is well formed. The following excerpt from the method checks the + and the − nodes:

```
if ((nodename == PlusNode) or (nodename == MinusNode)){
        Type1 = expression(T->get_child(0));
        if(T->get_degree()==2){
                Type2 = expression(T->get_child(1));
        }else{
                Type2 = TypeInteger;
        }
        if( (Type1 != TypeInteger) or (Type2 != TypeInteger)){
                error(T);
                cout << "ARGUMENTS OF '+', '-' etc. MUST BE OF TYPE\
                                                INTEGER" <<endl;
        }
        return TypeInteger;
}
```

To constrain the newly added "*" node, the following piece of code should be added to the method:

```
if (nodename == MultNode){
        Type1 = expression(T->get_child(0));
        Type2 = expression(T->get_child(1));

        if( (Type1 != TypeInteger) or (Type2 != TypeInteger)){
                error(T);
                cout << "ARGUMENTS OF '*', MUST BE OF TYPE\
                                                INTEGER" <<endl;
        }
        return TypeInteger;
}
```

**Changes to the Code Generator**

The code for arithmetic expressions is generated by a method *expression()* in the *tws::codegenerator* class. The following excerpt from the method shows how code is generated for the "-" node:

```
if (name == MinusNode){
        expression(T->get_child(0),CurrLabel);
        if (T->get_degree() == 2){
              expression(T->get_child(1),NoLabel);
              codegen(NoLabel,BOPOP,BMINUS);
              dec_framesize();
        }else{
              codegen(NoLabel,UOPOP,UNEG);
        }
 }
```

The code for the "*" node can be similarly generated as follows:

```
if (name == MultNode){
        expression(T->get_child(0),CurrLabel);
        expression(T->get_child(1),NoLabel);
        codegen(NoLabel,BOPOP,BMULT);
        dec_framesize();
 }
```

BMULT is the opcode for the multiplication operator in the target virtual machine used by this framework.

## Adding Statements

This example shows how a WHILE-DO statement can be added to Tiny. Two tokens while and do are being introduced which will affect the lexical analyzer. The parser specification will change to incorporate the new production and the constrainer and code generator will be modified appropriately. The details follow.

**Changes to the Lexical Analyzer – *lex.tiny***

The following rules are added to identify *repeat* and *until* as valid tokens :

```
"repeat"              { return rule(REPEAT); }
"until"               { return rule(UNTIL ); }
```

**Changes to the Parser – *parse.tiny***

The *Statement* production is modified to include the following new rule:

```
Statement   -> REPEAT Statement UNTIL Expression => "repeat";
```

The new *Statement* is as follows:

```
Statement  -> Name ASSIGNMENT Expression              => "assign"
           -> OUTPUT '(' Expression ')'               => "output"
           -> IF Expression THEN Statement
              ELSE Statement                          => "if"
           -> WHILE Expression DO Statement           => "while"
           -> REPEAT Statement UNTIL Expression       => "repeat"
           -> Body
           ->                                         => "<null>";
```

**Modifications to the Constrainer**

The tree node for the *while* statement is registered in *nodes.h* as

```
addnode(WhileNode,"while");
```

A new tree node for *repeat* is registered in *nodes.h* as

```
addnode(RepeatNode,"repeat");
```

The *tws::constrainer* class' *process()* method is modified to include the following code:

```
if (nodename == RepeatNode){
        //process the boolean expression
        if (expression(T->get_child(1)) != TypeBoolean){
              error(T);
              cout << "UNTIL EXPRESSION NOT OF TYPE BOOLEAN"<<endl;
              throw(0);
        }
        //process the body of the loop
        process(T->get_child(1));
 }
```

**Changes to the Code Generator**

The code generator will have to incorporate code that is very similar to the existing WHILE loop code:

```
    if (nodename == WhileNode){
            if (CurrLabel == NoLabel)
              Label1 = make_label();
            else Label1 = CurrLabel;
              Label2 = make_label();
            Label3 = make_label();
            expression(T->get_child(0), Label1);
            codegen(NoLabel,CONDOP,Label2,Label3);
            dec_framesize();
            codegen(Label2,GOTOOP,Label1,
                  process(T->get_child(1),Label2));
            return Label3;
}
```

The following code is added to the *process()* method of *tws::codegenerator* class to generate code for the REPEAT-UNTIL construct:

```
    if (nodename == RepeatNode){
            if (CurrLabel == NoLabel)
                  Label1 = make_label();
            else Label1 = CurrLabel;
            Label2 = make_label();
            std::string tmpLabel =
                  process(T->get_child(0),Label1);
            expression(T->get->child(1),tmpLabel);
            codegen(NoLabel,CONDOP,Label2,Label1);
            dec_framesize();
            return Label2;
     }
```

More complicated constructs like new data types, subroutines, enumerated data types, *for* loops, *case* statements etc. can be added to the language incrementally in a similar fashion.

## Conclusions and Future Work

The translator writing system in C++ achieves its goal of refactoring the constrainer and code generator out of a compiler writing tool like *bison* and facilitating incremental development of the system. This system can be used to teach translation in which every part of the compiler is repeatedly visited to add new constructs to the language.

# APPENDIX A
# THE PGEN META GRAMMAR

This appendix gives the listing of *Parser.ypp*, the *bison* input file which specifies the language in which programming language syntax is specified in the TWS.

```
%{
#include <iostream>
#include <cassert>
#include "Tokenizer.h"
#include "tree.h"
#include "Nonterminals.h"

extern tws::tree<TOKEN_INFO> *root;

void yyerror(char *err)
{
      std::cout<<err<<std::endl;
}
#define YYDEBUG 1
%}

%union{
 TOKEN_INFO info;
 tws::tree<TOKEN_INFO> *nodeptr;
}

%token <info> MARK NODE OR LIST ACTION IDENTIFIER LITERAL STRING
%token <info> LIT /* single chars like ';' */
%token <info> PIPE STAR PLUS QUESTION
%type  <nodeptr> spec defs def nlist yid rules rule or rbody
%type  <nodeptr> rexp term fact prim rop

%%

spec: defs MARK rules
          {
              TOKEN_INFO tok = make_token(t_spec,"spec");
              tws::tree<TOKEN_INFO>* t = new
tws::tree<TOKEN_INFO>(tok);
              t->add_children($1,$3,0);
              root = t;
          }
      ;

defs  : /* empty  */
          {
              TOKEN_INFO tok=make_token(t_defs,"defs");
```

```
              tws::tree<TOKEN_INFO>* t = new
tws::tree<TOKEN_INFO>(tok);
              $$ = t;
          }
      | defs def
          {
              $1->add_child($2);
            $$ = $1;
          }
      ;

def   : NODE nlist
          {
            $$ = $2;
          }
      ;

nlist : yid
          {
              tws::tree<TOKEN_INFO>* t = new
tws::tree<TOKEN_INFO>(make_token(t_def,"def"));
              t->add_child($1);
            $$ = t;
          }
      | nlist yid
          {
              $1->add_child($2);
            $$ = $1;
          }
      | nlist ',' yid
          {
              $1->add_child($3);
            $$ = $1;
          }
      ;

yid   : IDENTIFIER
          {
              TOKEN_INFO tok = make_token($1);
              tws::tree<TOKEN_INFO> *t = new
tws::tree<TOKEN_INFO>(tok);
            $$ = t;
          }
      | LITERAL
          {
              TOKEN_INFO tok = make_token($1);
              tws::tree<TOKEN_INFO>* t = new
tws::tree<TOKEN_INFO>(tok);
            $$ = t;
          }
      ;

/* rules section  */

rules : rule
          {
              TOKEN_INFO tok = make_token(t_rules,"rules");
```

```
              tws::tree<TOKEN_INFO>* t = new
tws::tree<TOKEN_INFO>(tok);
              t->add_child($1);
            $$ = t;
          }
      | rules rule
          {
              $1->add_child($2);
            $$ = $1;
          }
      ;

rule  : IDENTIFIER OR rbody ';'
          {
              tws::tree<TOKEN_INFO> *rule,*id,*Or;
              rule = new
tws::tree<TOKEN_INFO>(make_token(t_rule,"rule"));
              id   = new tws::tree<TOKEN_INFO>(make_token($1));
              Or   = new tws::tree<TOKEN_INFO>(make_token($2));
              Or->add_child($3);
              rule->add_children(id,Or,0);
            $$ = rule;
          }
      | IDENTIFIER OR rbody ACTION STRING ';'
          {
              tws::tree<TOKEN_INFO> *rule,*id,*action,*string,*Or;

              rule = new
tws::tree<TOKEN_INFO>(make_token(t_rule,"rule"));
            id   = new tws::tree<TOKEN_INFO>(make_token($1));
           Or   = new tws::tree<TOKEN_INFO>(make_token($2));
           action = new tws::tree<TOKEN_INFO>(make_token($4));
           string = new tws::tree<TOKEN_INFO>(make_token($5));
              Or->add_children($3,action,string,0);
              rule->add_children(id,Or,0);
            $$ = rule;
          }
      | IDENTIFIER OR rbody or ';'
          {
              tws::tree<TOKEN_INFO> *rule,*id,*Or,*t;

              rule = new
tws::tree<TOKEN_INFO>(make_token(t_rule,"rule"));
              id   = new tws::tree<TOKEN_INFO>(make_token($1));
              Or   = new tws::tree<TOKEN_INFO>(make_token($2));
              Or->add_child($3);
              rule->add_children(id,Or,$4,0);

              $4->lift();
              delete $4;
            $$ = rule;
        }
      | IDENTIFIER OR rbody ACTION STRING or ';'
          {
              tws::tree<TOKEN_INFO>* rule,*id,*Or,*action,*string,*t;
              rule = new
tws::tree<TOKEN_INFO>(make_token(t_rule,"rule"));
```

```
                id   = new tws::tree<TOKEN_INFO>(make_token($1));
                Or   = new tws::tree<TOKEN_INFO>(make_token($2));
                action = new tws::tree<TOKEN_INFO>(make_token($4));
                string = new tws::tree<TOKEN_INFO>(make_token($5));
                Or->add_children($3,action,string,0);
                rule->add_children(id,Or,$6,0);

                $6->lift();
                delete $6;
              $$ = rule;
          }
        ;

or    : OR rbody
            {
                tws::tree<TOKEN_INFO> *dummy,*Or;
                dummy = new
tws::tree<TOKEN_INFO>(make_token(t_dummy,"dummy"));
              Or    = new tws::tree<TOKEN_INFO>(make_token($1));
                Or->add_child($2);
                dummy->add_child(Or);
              $$ = dummy;
            }
        | OR rbody ACTION STRING
            {
                    tws::tree<TOKEN_INFO> *dummy,*Or,*action,*string;
                    dummy = new
tws::tree<TOKEN_INFO>(make_token(t_dummy,"dummy"));

                    Or    = new tws::tree<TOKEN_INFO>(make_token($1));
                    action= new tws::tree<TOKEN_INFO>(make_token($3));
                    string =new tws::tree<TOKEN_INFO>(make_token($4));
                    Or->add_children($2,action,string,0);
                    dummy->add_child(Or);
                $$ = dummy;
          }
        | or OR rbody
            {
                    tws::tree<TOKEN_INFO>*Or;
                    Or = new tws::tree<TOKEN_INFO>(make_token($2));
              Or->add_child($3);
                    $1->add_child(Or);
                    $$ = $1;
          }
        | or OR rbody ACTION STRING
            {
                    tws::tree<TOKEN_INFO> *Or,*action,*string;

                    Or = new tws::tree<TOKEN_INFO>(make_token($2));
                    action = new tws::tree<TOKEN_INFO>(make_token($4));
                    string = new tws::tree<TOKEN_INFO>(make_token($5));
                    Or->add_children($3,action,string,0);
                    $1->add_child(Or);
                    $$ = $1;
          }
        ;
```

```
rbody : /*      empty  */
          {
                   tws::tree<TOKEN_INFO>* t;
                   t = new
tws::tree<TOKEN_INFO>(make_token(t_empty,"empty"));
                   $$ = t;
          }
      | rexp  /* rbody yid */
          {
             $$ = $1;
          }
/* | rbody act */
      ;

rexp  : term
          {
                   $$ = $1;
          }
      | PIPE term
          {
                   tws::tree<TOKEN_INFO>* t;
                   t = new tws::tree<TOKEN_INFO>(make_token($1));
                   t->add_child($2);
                   $$ = t;
          }
      | rexp PIPE term
          {
                   tws::tree<TOKEN_INFO>* t;
                   t = new tws::tree<TOKEN_INFO>(make_token($2));
                   t->add_children($1,$3,0);
                   $$ = t;
          }
      ;

term  : fact
          {
                   $$ = $1;
          }
      | term fact
          {
                   tws::tree<TOKEN_INFO>* t;
                   t = new
tws::tree<TOKEN_INFO>(make_token(t_catenate,"catenate"));
                   t->add_children($1,$2,0);
                   $$ = t;
          }
      ;

fact  : prim
          {
                   $$ = $1;
          }
      | prim rop
          {
                   $2->add_child($1);
                   $$ = $2;
          }
```

```
      ;


prim  : yid
          {
                  $$ = $1;
          }
      | '(' rexp ')'
          {
                  $$ = $2;
          }
      ;

rop   : PLUS
          {
                  tws::tree<TOKEN_INFO>* t;
                  t = new tws::tree<TOKEN_INFO>(make_token($1));
                  $$ = t;
          }
      | STAR
          {
                  tws::tree<TOKEN_INFO>* t;
                  t = new tws::tree<TOKEN_INFO>(make_token($1));
                  $$ = t;
          }
      | QUESTION
          {
                  tws::tree<TOKEN_INFO>* t;
                  t = new tws::tree<TOKEN_INFO>(make_token($1));
                  $$ = t;
          }
      | LIST prim
          {
                  tws::tree<TOKEN_INFO>* t;
                  t = new tws::tree<TOKEN_INFO>(make_token($1));
                  t->add_child($2);
                  $$ = t;
          }
      ;
```

APPENDIX B
SOURCE CODE FOR *LEX.TINY* AND *PARSE.TINY*

**lex.tiny**

```
%option noyywrap
%{
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <Tokenizer.h>
#include "dlist.h"

static int line = 1;
static int column = 1;
int rule(int token);
int node(int token);
void yyerror(char* message);
int debug_tokenizer = 0;
typedef tws::dlist<tws::tree<TOKEN_INFO>*> dlist;

#include "code.tab.hpp"
%}
%x COMM1
COMM2   #.*\n
IDENT  [_a-zA-Z][_a-zA-Z0-9]*
INT    [0-9]+
WHITE   [ \t\v\f]*
LIT     '[^\']+'
STR     \"[^\"]+\"
%%

{WHITE}       { column += yyleng; }
\n            { column = 1; line++; }
"program"     { return rule(PROGRAM); }
"var"         { return rule(VAR); }
"integer"     { return rule(INTEGER); }
"boolean"     { return rule(BOOLEAN); }
"begin"       { return rule(BEGINX); }
"end"         { return rule(END); }
":="          { return rule(ASSIGNMENT); }
"output"      { return rule(OUTPUT); }
"if"          { return rule(IF); }
"then"              { return rule(THEN); }
"else"              { return rule(ELSE); }
"while"       { return rule(WHILE); }
"do"          { return rule(DO); }
"<="          { return rule(LTE); }
"read"              { return rule(READ); }
{INT}         { return node(INTEGER_NUM); }
{IDENT}             { return node(IDENTIFIER); }
"{"           { column += yyleng; BEGIN( COMM1 ); }
<COMM1>[^}\n]+  { column += yyleng; }
<COMM1>"}"    { column += yyleng; BEGIN( INITIAL ); }
<COMM1>\n     { column = 1; line++; }
```

```
{COMM2}         { column = 1; line++; }
":"             { return rule(yytext[0]); }
";"             { return rule(yytext[0]); }
"."             { return rule(yytext[0]); }
","             { return rule(yytext[0]); }
"("             { return rule(yytext[0]); }
")"             { return rule(yytext[0]); }
"+"             { return rule(yytext[0]); }
"-"             { return rule(yytext[0]); }
.               { yyerror("unrecognized char");
                  printf("-->%s<--\n",yytext);
                  column++; }
%%

int rule(int token)
{
    if (debug_tokenizer) {
       printf("string:%s, token: %d, line: %d, column: %d\n",
              yytext,token,line,column);
    }

    yylval.info.line = line;
    yylval.info.column = column;
    column += yyleng;

    yylval.info.string = (char*)malloc(yyleng+1);
    assert(yylval.info.string);
    strcpy(yylval.info.string, yytext);

    yylval.info.makenode = 0;

    return token;
}

int node(int token)
{
    int tok = rule(token);
    yylval.info.makenode = 1;
    return tok;
}
```

## parse.tiny

```
%%
Tiny       -> PROGRAM Name ':' Dclns Body Name '.'          => "program";
Dclns      -> VAR (Dcln ';')*                               => "dclns"
           ->                                               => "dclns";
Dcln       -> Name list ',' ':' Type                        => "dcln";
Type       -> INTEGER                                       => "integer"
           -> BOOLEAN                                       => "boolean";
Body       -> BEGINX Statement list ';' END                 => "block";
Statement  -> Name ASSIGNMENT Expression                    => "assign"
           -> OUTPUT '(' Expression ')'                     => "output"
           -> IF Expression THEN Statement
               ELSE Statement                               => "if"
           -> WHILE Expression DO Statement                 => "while"
           -> Body
            ->                                              => "<null>";

Expression -> Term
            -> Term LTE Term                                 => "<=";

Term       -> Primary
            -> Primary '+' Term                             => "+";

Primary    -> '-' Primary                                   => "-"
            -> READ                                         => "read"
           -> Name
           -> INTEGER_NUM                                   => "<integer>"
           -> '(' Expression ')';

Name       -> IDENTIFIER                                    => "<identifier>";
```

APPENDIX C
SOURCE CODE FOR THE INITIAL CONSTRAINER
AND CODE GENERATOR

**nodes.h**

```
#if defined (addnode)
      #undef addnode
#endif

#if defined(NODES_DEFINE_CONSTRAINER)
      #define addnode(A,B) std::string tws::constrainer::A = B;
#elif defined(NODES_DEFINE_CODEGENERATOR)
      #define addnode(A,B) std::string tws::cgen::A = B;
#elif defined(CONSTRAINER_H) || defined(CODEGEN_H)
      #define addnode(A,B) static std::string A;
#else
      #define addnode(A,B)
#endif

      addnode(ProgramNode,"program");
      addnode(TypesNode, "types");
      addnode(TypeNode, "type");
      addnode(DclnsNode ,"dclns");
      addnode(DclnNode, "dcln");
      addnode(IntegerTNode,"integer");
      addnode(BooleanTNode, "boolean");
      addnode(BlockNode,"block");
      addnode(AssignNode, "assign");
      addnode(OutputNode, "output");
      addnode(IfNode ,"if");
      addnode(WhileNode ,"while");
      addnode(NullNode , "null");
      addnode(LENode ,"<=");
      addnode(PlusNode ,"+");
      addnode(MinusNode,"-");
      addnode(ReadNode,"read");
      addnode(IntegerNode,"<integer>");
      addnode(IdentifierNode,"<identifier>");

      //ADD MORE NODES HERE
```

## tws::constrainer class

### constrainer.h

```
#ifndef CONSTRAINER_H
#define CONSTRAINER_H

#include <iostream>
#include <string>
#include <map>
#include <utility>
#include "Tokenizer.h"
#include "tree.h"
#include "scoped_symtable.h"

namespace tws{

class constrainer{
#include "nodes.h"

      typedef tree<TOKEN_INFO> *UserType,Tree;
      std::map<std::string,std::string> nodes;
      //std::map<std::string, Tree*>
      static tws::scoped_symtable<std::string,Tree*> dclTbl;
      Tree* T;

      static UserType TypeBoolean,TypeInteger;

      const char* filename;

      //void addnode(const char*,const char*);
      static Tree* last_child(Tree*);
      static const char* Name(Tree*);
public:

      constrainer(const char* filename);
      bool constrain();
      void addintrinsics();
      static void process(Tree* T);
      static void error(Tree* T);
      static UserType expression(Tree* T);
      //void init_nodes();
};

}
#endif
```

### constrainer.cpp

```
#include "constrain.h"
#include <fstream>
#include "NonTerminals.h"
```

```
using namespace std;
using namespace tws;



tws::constrainer::Tree* constrainer::last_child(tws::constrainer::Tree*
t)
{
      return t->get_child(t->get_degree()-1);
}

const char* tws::constrainer::Name(tws::constrainer::Tree* t)
{
      return t->get_data().string;
}


tws::scoped_symtable<std::string,tws::constrainer::Tree*>
tws::constrainer::dclTbl;
tws::constrainer::UserType tws::constrainer::TypeBoolean,
tws::constrainer::TypeInteger;

constrainer::constrainer(const char* filename):filename(filename)
{
      ifstream in(filename);
      T = Tree::read_with_decorations(in);
      in.close();
      //T->print();
//    init_nodes();
}



bool constrainer::constrain()
{
      addintrinsics();
      T->print();
      try{
            constrainer::process(T);
            ofstream out(filename);
            Tree::dump_with_decorations(out,T);
            out.close();
            return true;
      }catch(...){
            return false;
      }
}

void constrainer::addintrinsics()
{
      //add types node
      Tree* t = new Tree(make_token(t_dummy,"types"));
      T->insert_child(t,1);

      Tree* t1 = new Tree(make_token(t_dummy,"integer"));
      Tree* t2 = new Tree(make_token(t_dummy,"boolean"));
```

```
	Tree* p_t1 = new Tree(make_token(t_dummy,"type"));
	Tree* p_t2 = p_t1->clone();

	p_t1->add_child(t1);
	p_t2->add_child(t2);

	t->add_child(p_t2);
	t->add_child(p_t1);

	TypeBoolean = p_t2;
	TypeInteger = p_t1;
}

void constrainer::error(tws::constrainer::Tree* T)
{
	cout<<"<<< CONSTRAINER ERROR >>> AT"
		<< T->get_data()
		<< endl;
}

void constrainer::process(tws::constrainer::Tree* T)
{
	using namespace std;
	using namespace tws;

	string nodename = string(T->get_data().string);
	if(nodename == ProgramNode){
		dclTbl.open_scope();
		string
			name1 = T->get_child(0)->get_child(0)-
>get_data().string,
			name2 = T->get_child(T->get_degree()-1)-
>get_child(0)->get_data().string;
		if(name1 != name2){
			error(T);
			cout<<"PROGRAM NAMES DO NOT MATCH"<<endl;
			throw(0);
		}
		for(size_t i=1;i<T->get_degree()-1;i++){
			process(T->get_child(i));
		}
		dclTbl.close_scope();

	}else if (nodename == TypesNode){
		for(size_t i=0;i<T->get_degree();i++)
			process(T->get_child(i));

	}else if (nodename == TypeNode){
		string name = Name(T->get_child(0));
		dclTbl.insert(name,T);
	}else if (nodename == DclnsNode){
		for(size_t i=0;i<T->get_degree();i++)
			process(T->get_child(i));
	}else if (nodename == DclnNode){
		string name1 = Name(last_child(T));
		UserType Type1 = dclTbl.lookup(name1);
```

```
            for(size_t i=0;i<T->get_degree()-1;i++){
                  string name  = Name((T->get_child(i))->get_child(0));
                  dclTbl.insert(name,T->get_child(i));
                  T->get_child(i)->decorate(Type1);
            }
      }else if (nodename == BlockNode){
            for(size_t kid = 0; kid < T->get_degree(); kid++){
                  process(T->get_child(kid));
            }
      }else if (nodename == AssignNode){
            UserType
                  Type1 = expression(T->get_child(0)),
                  Type2 = expression(T->get_child(1));
            if (Type1 != Type2){
                  error(T);
                  cout << "ASSIGNMENT TYPES DO NOT MATCH" <<endl;
                  throw(0);
            }
      }else if (nodename == OutputNode){
            for(size_t kid = 0; kid < T->get_degree();kid++){
                  if(expression(T->get_child(kid)) != TypeInteger ){
                        error(T);
                        cout <<"OUTPUT EXPRESSION MUST BE TYPE INTEGER"
<< endl;
                        throw (0);
                  }
            }
      }else if (nodename == IfNode){
            if(expression(T->get_child(0)) != TypeBoolean){
                  error(T);
                  cout << "CONTROL EXPRESSION OF 'IF' STMT IS NOT TYPE
BOOLEAN" << endl;
            }
            process(T->get_child(1));
            if (T->get_degree() == 3)
                  process(T->get_child(2));

      }else if (nodename == WhileNode){
            if (expression(T->get_child(0)) != TypeBoolean){
                  error(T);
                  cout << "WHILE EXPRESSION NOT OF TYPE BOOLEAN"
<<endl;
                  throw(0);
            }
            process(T->get_child(1));
      }else if (nodename == NullNode){
            //do nothing
      }else {
            error(T);
            cout << "UNKNOWN NODE NAME: " ;
            cout << Name(T) << endl;
      }
}

tws::constrainer::UserType
tws::constrainer::expression(tws::constrainer::Tree* T)
{
```

```
      using namespace std;
      using namespace tws;

      UserType Type1,Type2;
      string nodename = string(T->get_data().string);

      if (nodename == LENode){
            Type1 = expression(T->get_child(0));
            Type2 = expression(T->get_child(1));
            if (Type1 != Type2){
                  error(T);
                  cout << "ARGUMENTS OF '<=' MUST BE TYPE INTEGER" <<
endl;
            }
            return TypeBoolean;
      }else if ((nodename == PlusNode) or (nodename == MinusNode)){
            Type1 = expression(T->get_child(0));
            if(T->get_degree()==2){
                  Type2 = expression(T->get_child(1));
            }else{
                  Type2 = TypeInteger;
            }
            if( (Type1 != TypeInteger) or (Type2 != TypeInteger)){
                  error(T);
                  cout << "ARGUMENTS OF '+', '-' etc. MUST BE OF TYPE
INTEGER" <<endl;
            }
            return TypeInteger;

      }else if (nodename == ReadNode){
            return TypeInteger;
      }else if (nodename == IntegerNode){
            return TypeInteger;
      }else if (nodename == IdentifierNode){
            Tree *Declaration;
            try{
                  Declaration = dclTbl.lookup(Name(T->get_child(0)));
                  T->decorate(Declaration);
                  return Declaration->decoration();
            }catch(...){
                  //lookup failed
                  return TypeInteger;
            }

      }else {
            error(T);
            cout <<"UNKNOWN NODE NAME" << Name(T) << endl;
      }
}


#define NODES_DEFINE_CONSTRAINER
      #include "nodes.h"
#undef NODES_DEFINE_CONSTRAINER


int main()
```

```
{
	constrainer c("_TREE");
	c.constrain();
}
```

## tws::codegenerator class

### codegen.h

```
#if !defined(CODEGENERATOR_H)
#define CODEGENERATOR_H

#include "cgen.h"

namespace tws{

      class codegenerator : public tws::cgen {
      public:
            static void expression(Tree* T,std::string CurrLabel);
            static std::string process(Tree* T, std::string CurrLabel);
            static void codegenerate();

      };
}
#endif
```

### codegen.cpp

```
#include "codegen.h"
#include <fstream>

using namespace std;
using namespace tws;


void tws::codegenerator::codegenerate()
{
      using namespace std;
      using namespace tws;

      ifstream in("_TREE");
      tree<TOKEN_INFO>* root =
tree<TOKEN_INFO>::read_with_decorations(in);
      in.close();

      root->print();

      std::string HaltLabel = process(root,codegenerator::NoLabel);
      codegen(HaltLabel,codegenerator::HALTOP);

      ofstream out("_CODE");
      cgen::dumpcode(out);
      out.close();
}

int main()
```

```
{
	codegenerator::codegenerate();
}

void tws::codegenerator::expression(tws::cgen::Tree* T,std::string
CurrLabel)
{
	size_t kid;
	std::string Label1;

	std::string name = Name(T);


	if ( (name == LENode) or (name == PlusNode)){
		expression(T->get_child(0),CurrLabel );
		expression(T->get_child(1),NoLabel);

		if (Name(T) == LENode)
			codegen(NoLabel,BOPOP,BLE);
		else codegen(NoLabel,BOPOP,BPLUS);
		dec_framesize();
	}else if (name == MinusNode){
		expression(T->get_child(0),CurrLabel);
		if (T->get_degree() == 2){
			expression(T->get_child(1),NoLabel);
			codegen(NoLabel,BOPOP,BMINUS);
			dec_framesize();
		}else{
			codegen(NoLabel,UOPOP,UNEG);
		}
	}else if (name == ReadNode){
		codegen(CurrLabel,SOSOP,OSINPUT);
		inc_framesize();
	}else if (name == IntegerNode){
		codegen(CurrLabel,LITOP,Name(T->get_child(0)));
		inc_framesize();
	}else if (name == IdentifierNode){
		reference(T,RightMode,CurrLabel);
	}else {
		//error
		error(T);
		cout << "<<< CODE GENERATOR >>> :UNKNOWN NODE NAME: " <<
Name(T) << endl;
	}

}

std::string tws::codegenerator::process(tws::cgen::Tree* T, std::string
CurrLabel)
{
	size_t kid;
	std::string Label1,Label2,Label3;

	std::string nodename = Name(T);

	if(nodename == ProgramNode){
		size_t degree = T->get_degree();
```

```
            CurrLabel = process(T->get_child(degree-3),CurrLabel);
            CurrLabel = process(T->get_child(degree-2),NoLabel);
            return CurrLabel;
      }else if (nodename == TypesNode){
            for(kid = 0; kid < T->get_degree(); kid++)
                  CurrLabel = process(T->get_child(kid),CurrLabel);
            return CurrLabel;
      }else if (nodename == TypeNode){
            return CurrLabel;
      }else if (nodename == DclnsNode){
            for(kid=0;kid<T->get_degree();kid++){
                  CurrLabel = process(T->get_child(kid),CurrLabel);
            }
            if (T->get_degree() > 0)
                  return NoLabel;
            else return CurrLabel;
      }else if (nodename == DclnNode){
            for(kid=0;kid<T->get_degree()-1;kid++){
                  if (kid != 0)
                        codegen(NoLabel,LITOP,toString(0));
                  else codegen(CurrLabel,LITOP,toString(0));
                  long num = make_address();
                  T->get_child(kid)->decorate(num);
                  inc_framesize();
            }
            return NoLabel;
      }else if (nodename == BlockNode){
            for(kid=0;kid<T->get_degree();kid++)
                  CurrLabel = process(T->get_child(kid),CurrLabel);

            return CurrLabel;
      }else if (nodename == AssignNode){
            expression(T->get_child(1),CurrLabel);
            reference(T->get_child(0),LeftMode,NoLabel);
            return NoLabel;
      }else if (nodename == OutputNode){
            expression(T->get_child(0),CurrLabel);
            codegen(NoLabel,SOSOP,OSOUTPUT);
            dec_framesize();
            for(kid = 1;kid < T->get_degree(); kid++){
                  expression(T->get_child(kid),NoLabel);
                  codegen(NoLabel,SOSOP,OSOUTPUT);
                  dec_framesize();
            }
            codegen(NoLabel,SOSOP,OSOUTPUTL);
            return NoLabel;
      }else if (nodename == IfNode){
            expression(T->get_child(0),CurrLabel);
            Label1 = make_label();
            Label2 = make_label();
            Label3 = make_label();
            codegen(NoLabel,CONDOP,Label1,Label2);
            dec_framesize();
            codegen(Label1,GOTOOP,Label3,process(T-
>get_child(1),Label1));
            if (T->get_degree() == 3){
                  codegen(Label2,NOP,process(T->get_child(2),Label2));
```

```
            }else{
                  codegen(Label2,NOP);
            }

            return Label3;
      }else if (nodename == WhileNode){
            if (CurrLabel == NoLabel)
                  Label1 = make_label();
            else Label1 = CurrLabel;
            Label2 = make_label();
            Label3 = make_label();
            expression(T->get_child(0), Label1);
            codegen(NoLabel,CONDOP,Label2,Label3);
            dec_framesize();
            codegen(Label2,GOTOOP,Label1,process(T-
>get_child(1),Label2));
            return Label3;
      }else if (nodename == NullNode){
            return CurrLabel;
      }else {
            //error
            error(T);
            cout << "<<< CODE GENERATOR >>>: UNKNOWN NODE NAME "<<
Name(T) << endl;
      }
}
```

LIST OF REFERENCES


1. Bermudez, Manuel E., A Modern Approach to Teaching Computer Translation, Proceedings of the SSGRR (Scuola Superiore G. Reiss Romoli) Winter International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine and Mobile Technologies on the Internet, L'Aquila, Italy, January 6-10, 2003.

2. Aho, A., Sethi, R., Ullman, J., Compilers: Principles, Techniques and Tools, Addison-Wesley Pub Co., Reading, MA,1986.

3. Bermudez M., COP4620 Translators, Unpublished course material, University of Florida, Gainesville, FL, Summer Semester 2002

4. Wikipedia – The Free Encyclopedia, http://www.wikipedia.org/wiki/CYK_algorithm. Last Access: May 13, 2003.

5. Paxson V., Flex, A Fast Scanner Generator, http://dinosaur.compilertools.net/flex/index.html, 1995. Last Access: May 13, 2003.

6. Donnelly, C., Stallman, R., Bison, The YACC-compatible Parser Generator, http://dinosaur.compilertools.net/bison/index.html, 1995. Last Access: May 13, 2003.

7. Levine, J., Mason, T., Brown, D., Lex & Yacc, O'Reilly & Associates, Sebastopol, CA, 1992.

8. Scott M., Programming Language Pragmatics, Morgan Kaufmann Publishers, San Francisco, CA, 2000.

9. Stroustrup B., The C++ Programming Language, Addison-Wesley Pub Co, Reading, MA, 2000.

## BIOGRAPHICAL SKETCH

Srivatsan Madhavan was born in Srirangam, India, in 1979. He received his Bachelor of Engineering degree in computer science and engineering from the University of Madras, India, in 2001.

Srivatsan joined the University of Florida, Gainesville, to pursue a master's degree in Computer and Information Science and Engineering in August 2001 and expects to complete his degree in August 2003.