PREPARED BY: ISMAIL EL AKBOURI

LR PARSER TINY LANGUAGE

LR Parser For Tiny Programming Language

STIPPLE UNLIMITED

JUN 2024

Table of Content

1 introduction

The parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.1, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. In the best case, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

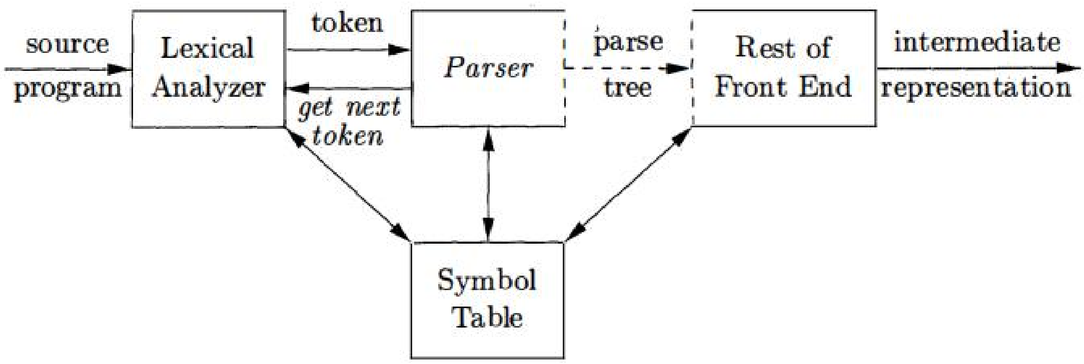

Figure 4.1: Position of parser in compiler model

The methods commonly used in compilers can be classified as being either top-down or bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-umethods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

The most efficient top-down and bottom-up methods work only for subclasses of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars; like, the predictive-parsing approach, which works for LL grammars. Parsers for the larger class of LR grammars are usually constructed using automated tools.

2 Context Free Grammar (CFG)

Context-free grammar (CFG) consists of a set of terminals, nonterminals, a start symbol, and productions rules.

- ✧ 1. Terminals are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal". The terminals are the keywords if and else and the symbols " c" and ") ."
- ✧ 2. Nonterminals are syntactic variables that denote sets of strings. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
- ✧ 3. In a grammar, one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
- ✧ 4. The productions rules: of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of:

    - ➢ A nonterminal called the head or left side of the production.
    - ➢ The symbol --+. Sometimes : : = has been used in place of the arrow.
    - ➢ A body or right side consisting of zero or more terminals and non terminals.

❖ **Notations**

Terminals	NonTerminals
- Lowercase alphabits <a,b,c,b,..> - Operators < +, * ,/,..> - Digites 0,1,2,3,4,..9 - Boldface string <if , then , id, ..>	- Supercase alphabits <A,B,C,...> - Lowercase string < *stm,exp*,..>

3 Error Handling

In the design and implementation of a parser for the Tiny programming language, error handling is a crucial component. The parser must efficiently detect, report, and recover from syntactic errors to assist programmers in locating and correcting mistakes in their code. Below is a summary of error handling strategies and their application in the Tiny parser.

3-1 Common Types of Errors

1. *Lexical Errors*: Misspellings of identifiers, keywords, or operators.
2. *Syntactic Errors*: Misplaced semicolons, missing braces, or invalid statement sequences.
3. *Semantic Errors*: Type mismatches between operators and operands.
4. *Logical Errors*: Incorrect reasoning or misuse of operators that result in incorrect program logic.

3-2 Error Detection

Parsing methods such as LL(1) and LR(1) are designed to detect errors efficiently, often as soon as an invalid token sequence is encountered. This is due to their *viable-prefix property*, which ensures that errors are detected as soon as the parser encounters a prefix of the input that cannot be completed to form a valid string in the language.

3-3 Error Recovery Strategies

- *Panic-Mode Recovery*: On detecting an error, the parser discards input tokens until it finds a designated synchronizing token

- *Phrase-Level Recovery*:The parser attempts to replace or insert tokens continue parsing

- *Error Productions:*The grammar includes special productions to handle common errors.

4 Tiny CFG

```
// These are the production rules which defined the Tiny programming
  language
// Consist of a set of terminals and set of nonteminals

--------------------------------------- Tiny CFG ---------------------------------------------

1.  program → stmt-sequence
2.  stmt-sequence → stmt-sequence ; statement
3.                  | statement
4.  statement → if-stmt
5.              | repeat-stmt
6.              | assign-stmt
7.              | read-stmt
8.              | write-stmt
9.  if-stmt → if exp then stmt-sequence end
10.           | if exp then stmt-sequence else stmt-sequence end
11. repeat-stmt → repeat stmt-sequence until exp
12. assign-stmt → id := exp
13. read-stmt → read id
14. write-stmt → write exp
15. exp → simple-exp relop simple-exp
16.       | simple-exp
17. simple-exp → simple-exp addop term
18.              | term
19. term → term mulop factor
20.        | factor
21. factor → (exp)
22.          | num
23.          | id
```

5 Parser Generator

While LL(1) parsers are commonly written by hand, LR(1) parsers are more challenging to do the same. Instead, we rely upon a parser generator to take a specification of a grammar and automatically produce the working code. during my project, I will give a shot to build a simple parser for Tiny language with Bison, a widely-used parser generator for C like languages. YACC (Yet Another Compiler Compiler) was a widely used parser generator in the Unix environment, recently supplanted by the GNU Bison parser which is generally compatible. Bison is designed to automatically invoke Flex as needed, so it is easy to combine the two into a complete program. Just as with the scanner, we must create a specification of the grammar to be parsed, where each rule followed by an action .

The overall structure of a Bison file is similar to that of Flex:

```
 %{
 #include<>
  #include<>      // C preamble code
 #include<>
 %}
        //declarations
%% ----------------------------------------------------------------------------------------------------
        // grammar rules

        // Example

expr : expr TOKEN_ADD expr
        | TOKEN_INT
        ;
%% ----------------------------------------------------------------------------------------------------
        // C postamble code

    int main()  {

     }
```

The first section contains arbitrary C code, typically #include statements and global declarations. The second section can contain a variety of declarations specific to the Bison language. We will use the %token keyword to declare all of the terminals in our language. In the example above indicating that non-terminal expr can produce the sentence expr TOKEN ADD expr or the single terminal TOKEN INT. White space is not significant, so it's ok to arrange the rules for clarity. Note that the usual naming convention is reversed: since upper case is customarily used for C constants, we use lower case to indicate non-terminals. The resulting code creates a single function yyparse() that returns an integer: zero indicates a successful parse, one indicates a parse error, and two indicates an internal problem such as memory exhaustion. yyparse assumes that there exists a function yylex that returns integer token types. This can be written by hand or generated automatically by Flex.You can see in fig1 the whole process of bothe lex and Bison

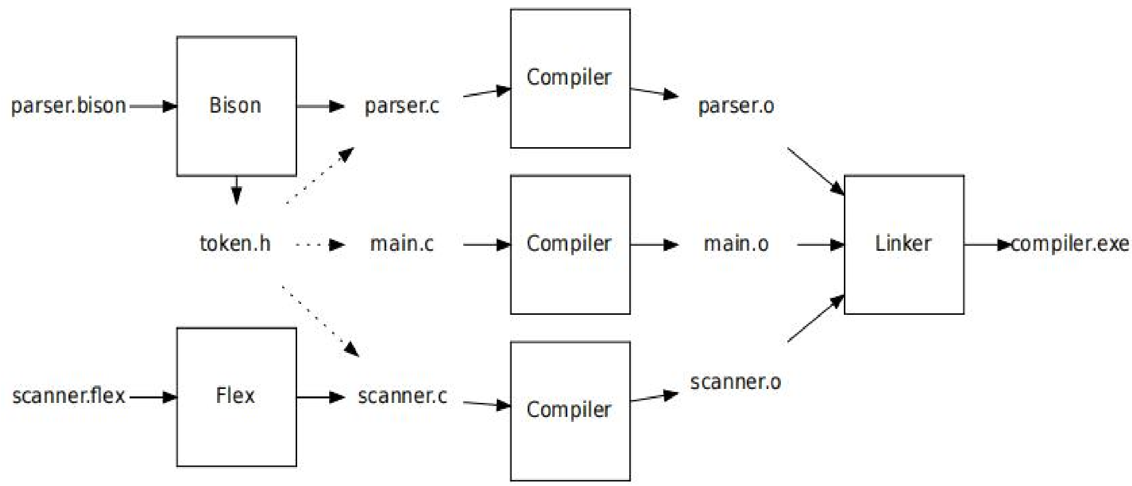

Fig1 : Procedure for Bison and Flex Together

6 Project structure

```
TParser/
├─── include/
|    └─── ast.h              //  header file to defined the ASR nodes
├─── lib/
|    ├─── setup.sh          // lex/yacc setup  shel script
|    └─── README.md   // a giude to start with lex / yacc tools
├─── out/
├─── src/
|    ├─── lexer.l            // lex file generator generate lex.yy.c
|    ├─── parser.y       // yacc file generator generate parser.yy.c and parser.tab.h
|    └─── ast.c         // functions to constructe and print AST
├─── tests/
|    ├─── test1.tiny
|    ├─── test2.tiny
|    ├─── test3.tiny       // tests files  test5 is incorrect tiny sample
|    ├─── test4.tiny
|    └─── test5.tiny
├─── Makefile           // Make file to build and manage the programs
```

7 Testing

7-1 Tiny samples

I choose multiple simple Tiny samples to test my parser, I mixed between samples which has some syntax error like test5.tiny and some which has unrecognized cahracters like test3 to show all possible cases ,but as I said this is just a simple version of Tiny parser maybe it can't parse more complex programs, and preceed the ast for them, but to do that you need to provide a complex and more generale Tiny CFG to yacc in order to generate a new upgraded vesion of Tiny parser

```
// Tiny sample 1

read x;
   if x > 0 then
     fact := 1;
 repeat
   fact := fact * x;
     x := x - 1
     until x = 0;
   write fact

end
```

```
// Tiny sample 2

   read a;
   read b;
     if a < b then
          c := b
     else
          c := a
end;
     write c
```

```
// Tiny sample 3

read x;
if x > 0 then
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
   @ until x = 0;
  write fact
end
```

7-2 AST samples

```
./out/tiny_parser < tests/test1.tiny
test parsing ...

Parsing completed successfully.

Abstract Syntax Tree:
=======================

1: Statement Sequence
├──> 2: Statement Sequence
│    └──> 3: Read: x
└──> 4: If Statement
│    ├──> 5: Relational Operator: >
│    │    ├──> 6: Factor: x (Identifier)
│    │    └──> 7: Factor: 0 (Number)
│    └──> 8: Statement Sequence
│    │    ├──> 9: Statement Sequence
│    │    │    ├──> 10: Statement Sequence
│    │    │    │    └──> 11: Assignment: fact
│    │    │    │    │    └──> 12: Factor: 0 (Number)
│    │    │    └──> 13: Repeat Statement
│    │    │    │    ├──> 14: Statement Sequence
│    │    │    │    │    ├──> 15: Statement Sequence
│    │    │    │    │    │    └──> 16: Assignment: fact
│    │    │    │    │    │    │    └──> 17: Multiplicative Operator: *
│    │    │    │    │    │    │    │    ├──> 18: Factor: fact (Identifier)
│    │    │    │    │    │    │    │    └──> 19: Factor: x (Identifier)
│    │    │    │    │    └──> 20: Assignment: x
│    │    │    │    │    │    └──> 21: Additive Operator: -
│    │    │    │    │    │    │    ├──> 22: Factor: x (Identifier)
│    │    │    │    │    │    │    └──> 23: Factor: 0 (Number)
│    │    │    │    └──> 24: Relational Operator: =
│    │    │    │    │    ├──> 25: Factor: x (Identifier)
│    │    │    │    │    └──> 26: Factor: 0 (Number)
│    │    └──> 27: Write
│    │    │    └──> 28: Factor: fact (Identifier)
```

```
./out/tiny_parser < tests/test2.tiny
test parsing ...

Parsing completed successfully.

Abstract Syntax Tree:
=======================

1: Statement Sequence
├──> 2: Statement Sequence
│    ├──> 3: Statement Sequence
│    │    ├──> 4: Statement Sequence
│    │    │    └──> 5: Read: a
│    │    └──> 6: Read: b
│    └──> 7: If Statement
│    │    ├──> 8: Relational Operator: <
│    │    │    ├──> 9: Factor: a (Identifier)
│    │    │    └──> 10: Factor: b (Identifier)
│    │    └──> 11: Statement Sequence
│    │    │    ├──> 12: Statement Sequence
│    │    │    │    └──> 13: Assignment: c
│    │    │    │    │    └──> 14: Factor: b (Identifier)
│    │    │    └──> 15: Statement Sequence
│    │    │    │    └──> 16: Assignment: c
│    │    │    │    │    └──> 17: Factor: a (Identifier)
└──> 18: Write
│    └──> 19: Factor: c (Identifier)
```

```
./out/tiny_parser < tests/test3.tiny
test parsing ...
Unrecognized character: @

Parsing completed successfully.

Abstract Syntax Tree:
=======================

1: Statement Sequence
├──> 2: Statement Sequence
│   └──> 3: Read: x
└──> 4: If Statement
│   ├──> 5: Relational Operator: >
│   │   ├──> 6: Factor: x (Identifier)
│   │   └──> 7: Factor: 0 (Number)
│   └──> 8: Statement Sequence
│   │   ├──> 9: Statement Sequence
│   │   │   ├──> 10: Statement Sequence
│   │   │   │   └──> 11: Assignment: fact
│   │   │   │   │   └──> 12: Factor: 0 (Number)
│   │   │   └──> 13: Repeat Statement
│   │   │   │   ├──> 14: Statement Sequence
│   │   │   │   │   ├──> 15: Statement Sequence
│   │   │   │   │   │   └──> 16: Assignment: fact
│   │   │   │   │   │   │   └──> 17: Multiplicative Operator: *
│   │   │   │   │   │   │   │   ├──> 18: Factor: fact (Identifier)
│   │   │   │   │   │   │   │   └──> 19: Factor: x (Identifier)
│   │   │   │   │   └──> 20: Assignment: x
│   │   │   │   │   │   └──> 21: Additive Operator: -
│   │   │   │   │   │   │   ├──> 22: Factor: x (Identifier)
│   │   │   │   │   │   │   └──> 23: Factor: 0 (Number)
│   │   │   │   └──> 24: Relational Operator: =
│   │   │   │   │   ├──> 25: Factor: x (Identifier)
│   │   │   │   │   └──> 26: Factor: 0 (Number)
│   │   └──> 27: Write
│   │   │   └──> 28: Factor: fact (Identifier)
```

8 Acknowledgments

This project was inspired by the LR parser implementation in the book "Compilers: Principles, Techniques, Tools " by Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman.
The parser generator tool Bison was used to generate the parser.
The lexical analyzer generator tool Flex was used to generate the lexer.
The CMake build system was used to manage the build process.
The C programming language was used to implement the parser and lexer.

9 References

Flex Manual:
https://westes.github.io/flex/manual/

GNU Bison Manual:
https://www.gnu.org/software/bison/manual/

Dragon Book Compilers Examples:
https://github.com/fool2fish/dragon-book-exercise-answers

TINY Language Specification:
https://www.cs.sjsu.edu/~mak/tutorials/TinyCompiler.pdf

For more related information please check the README.md file .

Made by: ISMAIL EL KABOURI
Subject: Tiny Language Parser