CS424

GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING AND TECHNOLOGY, TOPI, SWABI

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

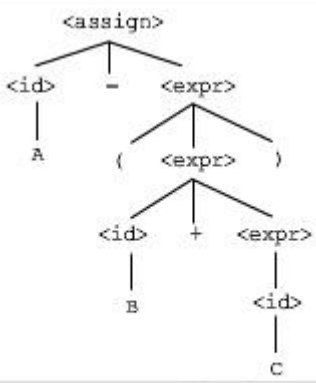
COMPILER
ASSIGNMENT 02

2020244 – MOHSIN ZIA

INSTRUCTOR: MR. USAMA ARSHAD
DATE: 11 MAR 2024

1. INTRODUCTION

A parser is like a detective. Its job is to look at a sentence (or a sequence of words) and figure out if it follows the rules of grammar, just like how a detective investigates a case to see if everything makes sense. For example, imagine you have a rule that says sentences in English must start with a capital letter and end with a period. A parser would check if a sentence does indeed start with a capital letter and end with a period. If it doesn't, the parser would say, "Hey, this doesn't follow the rules!"



Parse Tree for the Statement A = (B + C)

In programming, parsers do something similar but for code instead of sentences. They make sure that the code you write follows the rules of the programming language. If it doesn't, they'll let you know there's an error. This helps programmers catch mistakes early on and write better code.

There are different types of parsers, including:

1. **Recursive Descent Parser**: A top-down parser that recursively descends through the grammar rules to match the input.
2. **LR Parser (Left-to-Right, Rightmost Derivation)**: A bottom-up parser that constructs a parse tree in a left-to-right manner while performing a rightmost derivation. LR parsers are widely used in practice due to their efficiency and ability to handle a broad class of grammars.
3. **LL Parser (Left-to-Right, Leftmost Derivation)**: A top-down parser that constructs a parse tree in a left-to-right manner while performing a leftmost derivation.
4. **GLR Parser (Generalized LR Parser)**: An extension of LR parser that can handle ambiguous grammars by exploring multiple parse paths simultaneously.

2. MINILANG REQUIREMENTS

We've categorized our tokens into 8 different types to make them easier to understand. Here's what each type means:

- **DATATYPE**: They tell us what kind of information a variable holds. They can be things like "int" for whole numbers, "bool" for true or false values

- **KEYWORDS**: These are the special words that tell the computer to do something specific. For example, "print" tells the computer to show a message on the screen, "true" and "false" represent true or false values, and "if" and "else" help the computer make decisions.

- **IDENTIFIERS**: These are names we give to variables in our code. For instance, in the line "int Mohsin_123 = 123," "Mohsin_123" is the identifier.

- **LITERALS**: These are the actual pieces of information we give to our identifiers. In our program, we only accept values that are whole numbers (integers) or strings.

- **OPERATORS**: These symbols help us do math in our program. They include things like plus (+), minus (-), multiplication (*), and division (/).

- **PARANTHESIS**: These are tokens for symbols like "(" and ")", which help us group parts of our code together, especially in functions or statements.

- **COMMENTS**: The "//" is the comment sign.

- **OTHERS**: Anythings except these tokens are classified as others.

3. MINILANG IMPLEMENTATION

Firstly, I've created a lexical file in which I've defined the rules for each requirement and the action we want to perform on that pattern being matched.

```
/* Rule Section */
%%
[0-9]+ {
        yylval = atoi(yytext);
        return NUMBER;
    }
[a-zA-Z][a-zA-Z0-9]* {
        yylval = yylval = yytext[0]; // Just store the first character for now
        return VARIABLE;
    }
if { return IF; } // Rule for "if" keyword
else { return ELSE; } // Rule for "else" keyword
[\t] ;

[\n] { return 0; }

```

Then, I've created a yacc file in which we recognize the language minilang using regular expressions.

```
%token NUMBER
%token VARIABLE

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

/* Rule Section */
%%

Statement:
          | ArithmeticExpression { assignmentFlag = 0; }
          | Assignment { assignmentFlag = 1; }
          ;

ArithmeticExpression: E {
        printf("\nResult=%d\n", $$);
        return 0;
    };

E: E '+' E { $$ = $1 + $3; }
  | E '-' E { $$ = $1 - $3; }
  | E '*' E { $$ = $1 * $3; }
  | E '/' E { $$ = $1 / $3; }
  | E '%' E { $$ = $1 % $3; }
  | '(' E ')' { $$ = $2; }
  | NUMBER { $$ = $1; }
  | VARIABLE { $$ = variables[$1 - 'a']; } // Retrieve value of variable
  ;

Assignment: VARIABLE '=' E {
        variables[$1 - 'a'] = $3; // Assign value to variable
    };

%%
```

4. OUTPUT


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS C:\Users\Moshin\Desktop\CS424-Assignment_02> flex minilang.l
PS C:\Users\Moshin\Desktop\CS424-Assignment_02> bison -d minilang.y
PS C:\Users\Moshin\Desktop\CS424-Assignment_02> gcc .\lex.yy.c minilang.tab.c -w
PS C:\Users\Moshin\Desktop\CS424-Assignment_02> .\a.exe
Enter any arithmetic operator or variable assignment: a=9
Entered variable assignment is valid.
Enter any arithmetic operator or variable assignment: b=10
Entered variable assignment is valid.
Enter any arithmetic operator or variable assignment: 8+(8*9)
Result=80
Entered variable assignment is valid.
Enter any arithmetic operator or variable assignment: 80/8+20
Result=30
Entered variable assignment is valid.
Enter any arithmetic operator or variable assignment: 2+(40+9)*
Entered arithmetic expression is Invalid
Enter any arithmetic operator or variable assignment: