Compiler Design & Construction CS5900 CRN:11730

Group-25 Report

Team Members:

Seat No	Name	700#	ID
1	Mohammed Shariq Saaduddin Ahmed	700746549	mxa65490
2	Ganeshwara Annabathula	700759920	gxa99200
3	Sreeja Arava	700754019	sxa40190
4	Shuklavardhan Reddy Bandyala	700744486	sxb44860
5	Srujan Boddupalli	700756089	sxb60890

Programming Assignment:

P4) Write a yacc file without using NUMBER as token. You should not use scanf(). You should include: +, -, *, /, ( ).

P5) Write a yacc file. You may use scanf(), You should include: +, -, *, /, %, (-), ^, ( ).

P6) Write a yacc file. You may use scanf(). You should include: +, -, *, /, (-), ^, ( ).

About YACC

YACC

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar. (**Look-Ahead, Left-to-right, Rightmost Derivation parser**)
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

Structure of YACC

YACC Program consists of **3 sections**

1.Definition section

2.Rules & Action Section

3.Users Sub routine section

For Executing a YACC Program:

- **Save YACC File:** YACC programs are typically saved with a **.y** extension. YACC files contain the grammar rules and actions to be taken based on those rules.
- **Navigate to the Directory:** Using your command-line interface or terminal, you navigate to the directory where the .y file is saved.
- **Type yacc filename.y:** The **yacc** command is used to run the YACC program and generate a C-language file that contains the code for the parser. When you run **yacc filename.y**, YACC processes the grammar rules and generates a file usually named **y.tab.c**, which contains the C code for the parser based on the rules defined in **filename.y**.
- **Type gcc -lm y.tab.c:** The **gcc** command is a compiler used to compile C code. In this step, you use gcc to compile the **y.tab.c** file generated by YACC. The **-lm** flag links the math library, which might be necessary for certain mathematical functions used in your code.
- **Type ./a.out:** After successfully compiling the **y.tab.c** file, gcc generates an executable file named **a.out** by default. Typing **./a.out** in the terminal runs this executable, executing the program that was described by the rules in the YACC file.

Programming Assignment 4:

Definition Section

This section includes library imports and comments about the purpose of the Yacc program.

Precedence and Associativity of Operators

Precedence:

The higher precedence is given to the parenthesis '(' ')' following Multiplication '*' and division '/' have higher precedence than addition '+' and subtraction '-'. This precedence is reflected in the grammar structure. Operations involving * and / are handled before + and -.

Associativity:

Left-to-Right

Rules and Action Section

command: This is the starting rule for the grammar, indicating an expression. When the expression is parsed, it triggers the action of printing the result of the expression.

exp: Represents an expression. It can be a combination of terms either added or subtracted.

Rules within exp:

exp '+' term: Indicates addition of two terms.

exp '-' term: Indicates subtraction of two terms.

term: A single term.

term: Represents a term. It can be a combination of factors either multiplied or divided.

Rules within term:

term '*' factor: Indicates multiplication of two factors.

term '/' factor: Indicates division of two factors.

factor: A single factor.

factor: Represents a factor. It can be a number or an expression within parentheses.

Rules within factor:

number: A numerical value.

'(' exp ')': An expression within parentheses, allowing for grouping.

number: Represents a numerical value. It can be a single digit or multiple digits.

Rules within number:

number digit: Allows creating multi-digit numbers.

digit: Single digit values (0-9).

digit: Represents a single digit (0-9).

User Subroutine Section

main(): Initiates the parsing process by calling yyparse().

yylex(): Scans the input, discarding spaces, and returning characters until the end of the line or input is reached. Returns 0 at the end of input.

yyerror(char * s): Handles error messages if encountered during parsing.

How to make integers from digits

The "number" rule combines digits to form integers by calculating the value based on the digit sequence (with the base 10 principle), eventually forming complete numbers for calculations in the expression. The "digit" rule assigns numerical values to individual digits, and the "number" rule recursively combines these digits to form complete numbers.

Summary

The grammar rules define how expressions are structured and how to evaluate them based on arithmetic operations. The rules create a hierarchy where the most complex expressions (like multiplication and division) are built from simpler components (numbers, addition, subtraction) and ensure proper order of operations through the use of parentheses. The rules dictate how the different elements interact, guiding the parsing and calculation of arithmetic expressions.

Programming Assignment 5:

Definition Section

Similar to the previous program, it includes comments, defines the program's purpose, and imports necessary libraries (stdio.h, ctype.h, and math.h).

Precedence and Associativity of Operators

Precedence:

^ (Exponentiation)

*, /, %,(-) (Multiplication, Division, Modulo,Uminus)

+, - (Addition, Subtraction)

Associativity:

Right-associative for ^ (Exponentiation) and (-) Uminus

Left-associative for *, /, %, +, -

Rules and Action Section

command: Starting rule for the grammar, representing an expression. When parsed, it triggers the action of printing the result of the expression.

exp: Represents an expression, a combination of terms either added or subtracted.

Rules within exp:

exp '+' term: Indicates addition of two terms.

exp '-' term: Indicates subtraction of two terms.

term: A single term.

term: Represents a term, a combination of term2 elements either multiplied, divided, or used in modulo operations.

Rules within term:

term '*' term2: Indicates multiplication of two term2 elements.

term '/' term2: Indicates division of two term2 elements.

term '%' term2: Indicates modulo operation.

term2: A single term2.

term2: Represents a term with exponentiation.

Rules within term2:

factor '^' term2: Indicates exponentiation of two term2 elements.

factor: A single factor.

factor: Represents the basic components of the expression.

Rules within factor:

NUMBER: A numerical value.

('(' exp '): An expression within parentheses, allowing for grouping.

‘-‘ NUMBER: Negation of a number.

User Subroutine Section

main(): Initiates the parsing process by calling yyparse().

yylex(): Scans the input, identifying numbers, handling spaces, and returning characters until the end of the line or input is reached.

yyerror(char * s): Handles error messages if encountered during parsing.

Summary

This Yacc program extends the basic arithmetic operations by adding support for exponentiation, modulo and negation. It allows for the evaluation of more complex expressions by introducing features like exponentiation and handling negative numbers in the expression. The grammar rules establish the hierarchy of operations and enable the parsing and calculation of expressions involving these additional features.

Programming Assignment 6:

Definition Section

Similar to the previous programs, this section includes comments, defines the program's purpose, and imports necessary libraries (stdio.h, ctype.h, and math.h).

Precedence and Associativity of Operators

Precedence:

^ (Exponentiation)

*, /, %,(-) (Multiplication, Division, Modulo,Uminus)

+, - (Addition, Subtraction)

Associativity:

Right-associative for ^ (Exponentiation) and (-) Uminus

Left-associative for *, /, %, +, -

Additional Declarations

%token NUMBER: Identifies tokens as numbers.

%union {double val; char op;}: Declares a union type that can hold either a double value or a char representing an operator.

%type <val> exp term term2 factor NUMBER: Specifies the type associated with different elements of the grammar as <val>, indicating they will be of type double.

%type <op> '+'' '-' '*' '/' '(' ')' '^': Specifies the type associated with arithmetic operators as <op>, indicating they will be of type char.

Rules and Action Section

The grammar rules are similar to the previous program, enabling the calculation of expressions with added support for floating-point numbers.

factor: This rule now captures floating-point numbers.

NUMBER: {$$ = $1;} indicates that the token NUMBER will be a double.

yylex(): The lexer is modified to recognize floating-point numbers, as it checks for digits or a period (.) to identify floating-point input. It then scans the input as a double using scanf("%lf", &yylval).

"%lf" Format Specifier:

In C, %lf is used in the printf function to print double-precision floating-point numbers.

It's the format specifier for double data type in printf to output the value of a variable of type double.

Decimal Number Input:

The addition of (c == '.') condition in the yylex function allows the lexer to recognize and accept decimal numbers.

It reads characters and if it encounters either a digit or a decimal point, it recognizes it as a floating-point number.

Summary

This Yacc program is an extension of the previous integer calculator, allowing for calculations involving floating-point numbers. In this program we have removed the modulo operation and introducing support for floating-point values and modifying the lexer to recognize and handle these numbers, the grammar rules remain fundamentally the same while expanding the capability to perform calculations with decimal values.

Screenshots:

P4:

```
mxa65490@deepwater ~/cs5900 $ yacc group25_p4.y
mxa65490@deepwater ~/cs5900 $ gcc -lm y.tab.c
mxa65490@deepwater ~/cs5900 $ ./a.out
2+3
5
mxa65490@deepwater ~/cs5900 $ ./a.out
5-3
2
mxa65490@deepwater ~/cs5900 $ ./a.out
5*5
25
mxa65490@deepwater ~/cs5900 $ ./a.out
6/3
2
```

P5:

```
mxa65490@deepwater ~/cs5900 $ yacc group25_p5.y
mxa65490@deepwater ~/cs5900 $ gcc -lm y.tab.c
mxa65490@deepwater ~/cs5900 $ ./a.out
2^3^2
512
mxa65490@deepwater ~/cs5900 $ ./a.out
2*(3+6)
18
mxa65490@deepwater ~/cs5900 $ ./a.out
2%3+7
9
mxa65490@deepwater ~/cs5900 $ ./a.out
-2*3
-6
mxa65490@deepwater ~/cs5900 $ ./a.out
2^9^(-2)
1
```

P6:

```
mxa65490@deepwater ~/cs5900 $ yacc group25_p6.y
mxa65490@deepwater ~/cs5900 $ gcc -lm y.tab.c
mxa65490@deepwater ~/cs5900 $ ./a.out
.5*4
2.000000
mxa65490@deepwater ~/cs5900 $ ./a.out
5/2
2.500000
mxa65490@deepwater ~/cs5900 $ ./a.out
7/(2+3)
1.400000
```