Project 2: Yacc Programming

Introduction to Compilers (Fall 2024)

TA: Sanghyeon Lee (sanghyeon@snu.ac.kr)

Released: October 16, 2024
Due: October 30, 2024
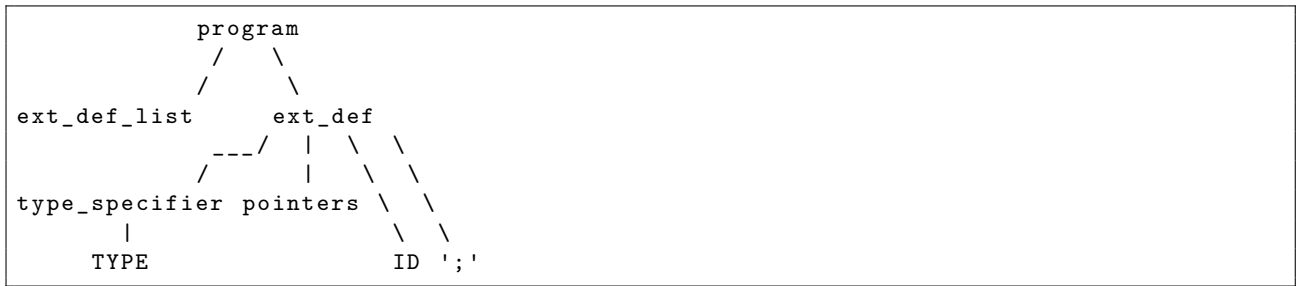
Contents

1 Introduction

1.1 Parsing

Syntactic analysis, also known as parsing, is the process of revealing the grammatical structure of a program by analyzing a sequence of tokens according to the language's grammar. This process is handled by the syntactic analyzer, also known as the parser. The parser accepts tokens analyzed by the lexer and finds a possible derivation in the grammar rules to build a structural representation of the program. This representation, known as the intermediate representation (IR), can take various forms such as a parse tree, an abstract syntax tree (AST), or intermediate code, which is widely adopted in modern compilers.

For example, consider the following code as an input program.

```
int a;
```

Then the lexer tokenizes it into `TYPE, ID, ';'`. The parser analyzes the structure of the program using these tokens. It finally builds a parse tree just like the following diagram.

```
           program
          /    \
         /      \
ext_def_list    ext_def
         ___/  |  \   \
        /      |   \   \
type_specifier pointers \   \
        |                 \   \
      TYPE               ID ';'
```

Since the syntactic analysis was successful, now we can determine that the input program is grammatically correct. In this project, we will build a parser that can accept and analyze programs conforming to the subC grammar.

1.2 Building Parsers

Parser generators can automatically build a parser with only a few lines of grammar rules. Similarly to lexers, there is no need to build a parser from scratch. GNU Bison is a widely used open-source LALR parser generator which is upward compatible with POSIX yacc (Yet Another Compiler-Compiler), a standard parser generator on Unix. Throughout this course, we will utilize Bison to generate parsers for subC.

2 Setting up the Project Environment

2.1 Pull the repository

Attach to the container and pull the repository to set up the environment for this project.

```
docker exec -it 2024-compilers bash
cd 2024-compilers
git pull
```

2.2 How to build and test

Navigate to the `src` directory, and build the project using `make` command.

```
cd ./src
make all
```

Once the build is completed, you can test it with the following command. If you do not specify the `[input_file]`, `stdin` will be used as the input stream.

```
./subc [input_file]
```

3 Different Features of subC from the Standard C

- No `typedef`
- No `union`
- No multi-dimensional array
- No pointer to a function
- No `long`, `short`, `float`, `double`, `unsigned`
- Only `int`, `char` and `void` built-in data types
- Allow nested comments (not a parsing issue).
- No explicit type conversion
- Strong type checking (semantic analysis issue, not a parsing one)

This section demonstrates the structure of our toy language, subC.

4 Grammar of subC

This section demonstrates the incomplete grammar of our toy language, subC.

Unlike project #1, please note that `float` *and* `..` *are removed in this project. Float constants are also deprecated.*

4.1 Symbols

The words in capitals represent terminals from the lexer.

Symbol Name	Symbol
`TYPE`	`int`, `char`
`VOID`	`void`
`STRUCT`	`struct`
`NULL`	`NULL`
`RETURN`	`return`
`INTEGER_CONST`	an integer constant
`CHAR_CONST`	a character constant
`STRING`	a string constant
`IF`	`if`
`ELSE`	`else`
`WHILE`	`while`
`FOR`	`for`
`BREAK`	`break`
`CONTINUE`	`continue`
`LOGICAL_OR`	`||`
`LOGICAL_AND`	`&&`
`RELOP`	`<, <=, >, >=`
`EQUOP`	`==, !=`
`INCOP`	`++`
`DECOP`	`--`
`STRUCTOP`	`->`

Table 1: Symbols in subC Grammar

4.2 Operator Precedence and Associativity

The operators assigned with the same precedence level are evaluated with the equal precedence, even if they are split across multiple rows.

Precedence	Operator	Name	Associativity
1	`++ --`	Suffix increment/decrement	Left
	`( )`	Function call and grouping	
	`.`	Structure member access	
	`->`	Structure member access through pointer	
2	`++ --`	Prefix increment/decrement	Right
	`-`	Unary minus	
	`!`	Logical NOT	
	`*`	Indirection (dereference)	
	`&`	Address-of	
3	`* / %`	Multiplication, division, and remainder	Left
4	`+ -`	Addition and subtraction	Left
5	`< <=`	Less-than and less-than-or-equal	Left
	`> >=`	Greater-than and greater-than-or-equal	
6	`== !=`	Equal and not-equal	Left
7	`&&`	Logical AND	Left
8	`||`	Logical OR	Left
9	`=`	Assignment	Right
10	`,`	Comma	Left

Table 2: Operator Precedence and Associativity

4.3   Productions

The following productions are written in the syntax of Bison grammar rules.

```
program
  : ext_def_list
  ;

ext_def_list
  : ext_def_list ext_def
  | %empty
  ;

ext_def
  : type_specifier pointers ID ';'
  | type_specifier pointers ID '[' INTEGER_CONST ']' ';'
  | func_decl compound_stmt
  ;

type_specifier
  : TYPE
  | VOID
  | struct_specifier
  ;

struct_specifier
  : STRUCT ID '{' def_list '}'
  | STRUCT ID
  ;

func_decl
  : type_specifier pointers ID '(' ')'
  | type_specifier pointers ID '(' VOID ')'
  | type_specifier pointers ID '(' param_list ')'
  ;

pointers
  : '*'
  | %empty
  ;

/* list of formal parameter declaration */
param_list
  : param_decl
  | param_list ',' param_decl
  ;

 /* formal parameter declaration */
param_decl
  : type_specifier pointers ID
  | type_specifier pointers ID '[' INTEGER_CONST ']'
  ;

/* list of definitions , definition can be type( struct ), variable , function */
def_list
  : def_list def
  | %empty
  ;

def
  : type_specifier pointers ID ';'
  | type_specifier pointers ID '[' INTEGER_CONST ']' ';'
  ;

compound_stmt
  : '{' def_list stmt_list '}'
  ;
```

```
stmt_list
  : stmt_list stmt
  | %empty
  ;

stmt
  : expr ';'
  | compound_stmt
  | RETURN ';'
  | RETURN expr ';'
  | ';'
  | IF '(' expr ')' stmt
  | IF '(' expr ')' stmt ELSE stmt
  | WHILE '(' expr ')' stmt
  | FOR '(' expr_e ';' expr_e ';' expr_e ')' stmt
  | BREAK ';'
  | CONTINUE ';'
  ;

expr_e
  : expr
  | %empty
  ;

expr
  : unary '=' expr
  | binary
  ;

binary
  : binary RELOP binary
  | binary EQUOP binary
  | binary '+' binary
  | binary '-' binary
  | binary '*' binary
  | binary '/' binary
  | binary '%' binary
  | unary %prec '='
  | binary LOGICAL_AND binary
  | binary LOGICAL_OR binary
  ;

unary
  : '(' expr ')'
  | '(' unary ')'
  | INTEGER_CONST
  | CHAR_CONST
  | STRING
  | ID
  | '-' unary %prec '!'
  | '!' unary
  | unary INCOP
  | unary DECOP
  | INCOP unary
  | DECOP unary
  | '&' unary
  | '*' unary %prec '!'
  | unary '[' expr ']'
  | unary '.' ID
  | unary STRUCTOP ID
  | unary '(' args ')'
  | unary '(' ')'
  ;

 /* actual parameters(function arguments) transferred to function */
```

```
args
  : expr
  | args ',' expr
  ;
```

5 Goal of the Project

You have to build a parser using Bison, which accepts the subC grammar and prints the production at each reduction. You can reuse the lexer and the hash table built in the previous project with some modifications, or create a new one for this project.

Complete the code in `subc.l`, `subc.y`, and `hash.c`. You may add or modify any files in the `src` directory as needed, but please make sure that the project can be built using `make all` command.

The provided grammar is incomplete, which means that there are some conflicts. Carefully inspect and resolve those conflicts. By adding the `-Wcounterexamples` flag, Bison will provide some counterexamples that can help you resolve conflicts. In the report, briefly describe the type of each conflict (reduce-reduce/shift-reduce) and how you resolved it in two pages or less.

5.1 Output format

At each reduction, print the grammar rule used for reduction to stdout. A demonstrative example of an input-output pair is given below.

Disregarding the output rules will result in a 5% penalty to your project score.

I/O Example

Input code: `./examples/ex_format/test.c`

```
int main(){
  int a;
  char b;

  a = 10;
  b = 5;

  if (a==10 || b==5) {
    return 1;
  } else {
    return 0;
  }
}
```

Output example: `./examples/ex_format/test_result`

```
$ ./subc ../examples/ex_format/test.c
ext_def_list->epsilon
type_specifier->TYPE
pointers->epsilon
func_decl->type_specifier pointers ID '(' ')'
def_list->epsilon
type_specifier->TYPE
pointers->epsilon
def->type_specifier pointers ID ';'
def_list->def_list def
type_specifier->TYPE
pointers->epsilon
def->type_specifier pointers ID ';'
def_list->def_list def
stmt_list->epsilon
unary->ID
unary->INTEGER_CONST
binary->unary
expr->binary
```

```
expr ->unary '=' expr
stmt ->expr ';'
stmt_list ->stmt_list stmt
unary ->ID
unary ->INTEGER_CONST
binary ->unary
expr ->binary
expr ->unary '=' expr
stmt ->expr ';'
stmt_list ->stmt_list stmt
unary ->ID
binary ->unary
unary ->INTEGER_CONST
binary ->unary
binary ->binary EQUOP binary
unary ->ID
binary ->unary
unary ->INTEGER_CONST
binary ->unary
binary ->binary EQUOP binary
binary ->binary LOGICAL_OR binary
expr ->binary
def_list ->epsilon
stmt_list ->epsilon
unary ->INTEGER_CONST
binary ->unary
expr ->binary
stmt ->RETURN expr ';'
stmt_list ->stmt_list stmt
compound_stmt ->'{' def_list stmt_list '}'
stmt ->compound_stmt
def_list ->epsilon
stmt_list ->epsilon
unary ->INTEGER_CONST
binary ->unary
expr ->binary
stmt ->RETURN expr ';'
stmt_list ->stmt_list stmt
compound_stmt ->'{' def_list stmt_list '}'
stmt ->compound_stmt
stmt ->IF '(' expr ')' stmt ELSE stmt
stmt_list ->stmt_list stmt
compound_stmt ->'{' def_list stmt_list '}'
ext_def ->func_decl compound_stmt
ext_def_list ->ext_def_list ext_def
program ->ext_def_list
```

6   Submission

- Place your report somewhere in the project directory. The file name must be `report.pdf`.

- Run `./submit.sh <student_number>` in the container. For example, `./submit.sh 1234-56789`. It will compress `src` directory and `report.pdf`.

- Submit the compressed archive on eTL.

- Please make sure that the name of the archive is `project2_<student_number>.zip` and your student number is correct. **Incorrect student number may result in a penalty for your project score.**

- For delayed submissions, there will be a 20% deduction in scores per day.

Appendix

GNU Bison Manual

[Yash (VS Code extension for Flex/Bison syntax highlighting)](#)