CS-1319: PLDI - Monsoon 23 **Team Name:** julius-stabs-back

Assignment #3

Instructor: PPD **Name:** Gautam Ahuja, Nistha Singh

Note: All of the codes are compiled on `Windows Subsystem for Linux version: 1.2.5.0` `using Ubuntu 22.04.2 LTS` over gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, flex 2.6.4 and bison (GNU Bison) 3.8.2.

Main C File: 3_A3.c File

Our main `c` file looks as follows:

```c
/* Group 03: julius-stabs-back
   Gautam Ahuja, Nistha Singh */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "3_A3.tab.h"

extern void yyerror(char *s);
extern int yyparse(void);

int main(){
    yyparse();
    return 0;
}
```

We include the `3_A3.tab.h` header which will be generated by the `--defines` flag in `bison`. We declare the two external functions `yyerror` and `yyparse` defined in the header file.
The call to `yyparse();` is where the parser starts.

Makefile

The `Makefile` consist of a rule for `build` as required in the assignment. It also consist of a rule `test` to test the parser on `3_A3.nc` file. Rest of the rules are for removing files depending on the requirements.

```
build:
	bison 3_A3.y --defines=3_A3.tab.h -o 3_A3.tab.c
	flex -o lex.yy.c 3_A3.l
	gcc -o parser lex.yy.c 3_A3.tab.c 3_A3.c -lfl -Werror

```

```
clean-head:
	rm -f lex.yy.c 3_A3.tab.c 3_A3.tab.h

clean-out: clean-head
	rm -f parser

test: build
	./parser < 3_A3.nc
```

There are a few more rules in the submitted `Makefile` but those are just to help debug.

Lexer: 3_A3.l

We used the same lexer as in Assignment 2. However there were a few changes as follows:

1. Error Correction
   We corrected two rules as those caused a test failure in the last assignment. We did a kleene closure on both rules instead of a positive closure.

   ```
   CHAR_SEQUENCE        {C_CHAR}*
   S_CHAR_SEQUENCE      {S_CHAR}*
   ```

2. Integer Constant
   The sign rule was removed form the integer constant lexer rule. Now it is as follows:

   ```
   NONZERO_DIGIT        [1-9]
   DIGIT                [0-9]
   INTEGER_CONSTANT     0|{NONZERO_DIGIT}{DIGIT}*
   ```

3. Return Statements
   Instead of printing the rules in `Definitions of Rules & Actions` section of lexer, we are returning them. The idea is when lexer captures a rule, it sends it to the parser. The parser then recognizes it as token (terminal symbol). The `return` statements are as follows:

   ```
   {CHARACTER}          { return CHARACTER; }
   {ELSE}               { return ELSE; }
   {FOR}                { return FOR; }
   {IF}                 { return IF; }
   {INTEGER}            { return INTEGER; }
   {RETURN}             { return RETURN; }
   {VOID}               { return VOID; }
   {IDENTIFIER}         { return IDENTIFIER; }
   .
   .
   .
   ```

4. PUNCTUATORS
   Since each of the individual punctuators are a terminal symbol as well, we need to differentiate them as the {PUNCTUATORS} rule returns all of them as same. In the parser, we can match the single punctuators with quotes ('?', '+', '*', etc.) This technique does not work for punctuators with more than one character (&&, $->$, etc.)
   We match each of the punctuators individually and return their token onto the parser. This way it becomes easier to handle.

```
"["                     { return L_BOX_BRACKET; }
"]"                     { return R_BOX_BRACKET; }
"("                     { return L_PARENTHESIS; }
")"                     { return R_PARENTHESIS; }
"{"                     { return L_CURLY_BRACE; }
"}"                     { return R_CURLY_BRACE; }
"->"                    { return ARROW; }
"&"                     { return AMPERSAND; }
"*"                     { return ASTERISK; }
"+"                     { return PLUS; }
.
.
.
```

5. Declarations
   Finally we add the `#include "3_A3.tab.h"` in declaration section of the lexer file. This is done to include the content of the `"3_A3.tab.h"` header file within the Flex file. This header also contains definitions for the tokens that the parser uses and thus the lexer is able to generate tokens accessible to parser.

```
/* Declarations */
%{
    #include <string.h>
    #include <stdio.h>
    #include "3_A3.tab.h"
%}
```

Bison Parser: 3_A3.y

We make the Bison Parser following the rules given in the assignment. We also follow the slides to generate the boiler-plate code for the parser as follows:

1. **Declarations:**
   We add the required header and function declarations in this part of the parser, similar to lexer.
   The function `yylex();` is declared here and accepts the tokens form lexer.

```
%{
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>
    extern int yylex();     // Lexical Analyzer generated by Flex
    void yyerror(char *s);  // Error function for Bison
    extern char* yytext;    // yytext declaration
%}
```

2. **Union:**
   A union is defined. This stores all the data types which are associated with token values. A token being parsed (eg. IDENTIFIER) will have a associated value with it when passed by the lexer. The datatype (`char *str`) for this is defined in union. This will be helpful when filling out the symbol table and performing semantic analysis.

```
%union {
    int intval;
    char* str;
};
```

3. **Tokens and Types:**
   The terminal symbols of grammar are denoted by `%token` and the non-terminals are denoted by `%type`. Although we do not explicitly need to define `%type`. The tokens are as follows:

```
/* Terminals */
%token CHARACTER                        // "char"
%token ELSE                             // "else"
%token FOR                              // "for"
%token INTEGER                          // "int"
.
.
.
%token <str> IDENTIFIER
%token <intval> INTEGER_CONSTANT
%token <str> CHARACTER_CONSTANT
%token <str> STRING_LITERAL
```

4. **Start Symbol:**
   By default the first is taken as a start rule. But since here we are using `translation_unit` as starting and it is not at initial position, we need to tell the lexer to start at this rule using `%start`.

```
/* start symbol */
%start translation_unit
```

5. **Associativity and Precedence Rule:**
   Though not needed, as the rules themselves are defined in a way to handle the Associativity and Precedence, we have still defined.

```
/* Operators Associativity and Precedence */
/* As per the slides of module 5 */
%right ASSIGN
%right QUESTION COLON
%left LOGICAL_OR
%left LOGICAL_AND
%left NOT_EQUAL IS_EQUAL
%left LESS_THAN LESS_THAN_EQUAL GREATER_THAN GREATER_THAN_EQUAL
%left PLUS MINUS
%left ASTERISK DIV MOD
```

6. **Grammar Rule:**
   We write all grammar rules as per assignment. We also add print statements as in guide.

```
unary_operator : AMPERSAND {printf("unary-operator\n");}
               | ASTERISK {printf("unary-operator\n");}
               | PLUS {printf("unary-operator\n");}
               | MINUS {printf("unary-operator\n");}
               | EXCLAMATION {printf("unary-operator\n");}
               ;
```

   We also define new rule for any rule having a `-opt` as follows:

```
argument_expression_list_opt : argument_expression_list
                             |
                             ;
```

7. **Error Function:**
   We define the error function as given in the guide.

```
void yyerror(char *s) {
    printf("Error: %s on '%s'\n", s, yytext);
}
```

Test File: 3_A3.nc

We used the following `3_A3.nc` file to test the parser.

```c
/* Group 03: julius-stabs-back */
/* Gautam Ahuja, Nistha Singh */

char* ArrayGrades(int *source, int *destination, int length) {
    int i;
        for (i = 0; i < length; i=i + 1) {
        destination[i] = source[i];
    }
        return destination;

        int a = source[0];
        int b = source[1];
        int c = source[2];
        if (a > b) {
        if (a > c) {
            return "Go";
        } else {
            return "Touch";
        }
    } else {
        if (b > c) {
            return "Some";
        } else {
            return "Grass";
        }
    }
        return "Hello, World! Seriously, what is this?";
}
```