CS323 - Compilers Project Phase 1

Basic features

Parser tree

First we need to set the type of `yylval` of our defined `ParserTreeNode`, notice that we use a union structure to record the different values of different terminals. For example, `int` is used to record the value of `INT` while `LITERAL` uses `char *`

```c
typedef struct ParserNode{
    char name[20];
    int line;
    int to_print_lineno;
    int child_num;
    int empty_value;
    struct ParserNode *child[10];
    union parser_node_value
    {
        int int_value;
        float float_value;
        char* string_value;
    } value;

} ParserNode;
```

We use a linked tree structure to record the parser tree.

For any derivation, we link an edge which uses variadic function.

Here is an example

```
Stmt: WHILE LP Exp RP Stmt { $$ = initParserNode("Stmt", yylineno);
addParserDerivation($$, $1, $2, $3, $4, $5, NULL); }

void addParserDerivation(struct ParserNode *node, ...) {
        va_list args;
        va_start(args, node);
        while(1) {
            struct ParserNode *child = va_arg(args, struct ParserNode*);
            if(child == NULL) break;
            addParserNode(node, child);
        }
        va_end(args);
    }
```

Lexical error

We match illegal tokens. The implementation is as follows.
We mainly detect lexical starting with illegal characters like a decimal followed by a bunch of letters, and also other characters like '@'.

```
illegal_id ({int}|{undefined_symbol}){identifier}
undefined_symbol (\'.{3,}\')|($)|(@)
...
{illegal_id} {  extern int lexeme_error; lexeme_error = 1;
                printf("Error type A at Line %d: unknown lexeme %s\n",
yylineno, yytext);
                PROCESS_TOKEN(TOKEN_ID) }
...
```

Syntax error

As required, we find syntax errors and do error recovery as far as we can. As a result, we pass all given basic test cases, many of which contain various syntax error detections. The syntax error we implemented is summarized below.

- Missing semicolon ';'
- Missing closing parenthesis ')'
- Missing specifier

As for implementation, we simply place the error token in a correct context for the first two case. But for the last one, we soon found that we had to add incorrect productions into the syntax specification. And in this process, we realize its difficulties. Any incorrect syntax addition will lead to a failure caused by the destruction of the original LALR parsing. The syntax specifications added are as follows:

Missing semicolon ';'

- ExtDef -> Specifier error
- ExtDef -> Specifier ExtDecList error
- Stmt -> Exp error
- Stmt -> RETURN Exp error
- Def -> Specifier DecList error

Missing closing parenthesis ')'

- FunDec -> ID LP error
- Stmt -> IF LP error
- Stmt -> FOR LP error
- Exp -> LP error
- Exp -> ID LP error

Missing specifier

- StmtList -> Stmt Def StmtList

Parse tree

We use a parse node struct to manage tree nodes in a unified manner. In addition, we implement several functions associated with it, which can be used in bison to help easily build tree structures. The implementation is as follows:

```
typedef struct ParserNode{
    char name[20];
    int line;
    int to_print_lineno;
    int child_num;
    int empty_value;
    struct ParserNode *child[10];
    union parser_node_value
    {
        int int_value;
        float float_value;
        char* string_value;
    } value;
} ParserNode;

// some more functions
void addParserNode(struct ParserNode *node, struct ParserNode *child);
ParserNode* initParserNode(const char *name, int lineno);
void cal_line(struct ParserNode *node);
void printParserNode(struct ParserNode *node, int depth);
```

Extended features

for statement

We simple add some syntax specification into our bison implementation, the detail is as follow:

- Stmt -> FOR LP Exp SEMI Exp SEMI Exp RP Stmt
- Stmt -> FOR LP Def Exp SEMI Exp RP Stmt

These two syntax can cover two different situations of for statement:

- for (int i = 0; 3 < 1; i = i + 1)
- for (declared_variable = 0; declared_variable < 3 ; declared_variable = declared_variable+1)

Our test case for this part is as follows:

```
int main() {
    for (int i = 1; i = 2; i = 3) {
        for (x = 5; x = 1; x = 2) {
            x = 5;
        }
    }
}
```

file inclusion

We use the file stream management in Flex to implement file inclusion.

Whenever we find an inclusion in the text, we create a new stream for the included file, and push it into the buffer stack in Flex, the core part of the code is as follow:

```
yyin = fopen(addr, "r");
yypush_buffer_state(yy_create_buffer(yyin, YY_BUF_SIZE));
free(addr);
BEGIN(INITIAL);
```

And after analyzing one file, we pop the current buffer out of the stack, and continue to work with the buffer at the top of the stack.

```
<<EOF>> {
    yypop_buffer_state();
    if (!YY_CURRENT_BUFFER) {
        yyterminate();
    }
 }
```

comment

We match two types of the comment in Flex and ignore them.

```
"//".* { }
"/*"(.|\n)*?"*/" { }
```