SPLCompiler

Team Members

11812102 Weijie Huang

11810517 Zhaoqi Xiao

11810206 Tao Sun

Design Idea

Lexical Analysis

We use Flex which is a fast lexical analyzer generator. We firstly specify the token patterns to match and actions to apply for each token. Flex can take my specification and generate a combined NFA that recognizes all the patterns, then convert it to an equivalent DFA, minimize the automaton as much as possible, finally generate C code that implements the lexer. Hence we will use Flex in the project to do Lexical Analysis.

Syntax Analysis

We use Bison which is a parser generator. we provide the input of a grammar specification and Bison will generate an LALR parser to recognize sentences in that grammar. Bison generates a parser, which accepts the input token stream from Flex, to recognize code in the specified context-free grammar (CFG).

Implementation

Parse Tree Construction

To build a parse tree, we define a struct and corresponding methods in `splc.l` , and use these methods in `splc.y`.

The key code in `splc.l`:

```
typedef struct tree_node tree_node;
typedef struct child_list_node child_list_node;

tree_node* make_tree_node(const char*, int, int);
child_list_node* make_child_list_node(tree_node*);
void add_child(tree_node*, tree_node*);
void show_tree(tree_node*, int);

struct tree_node {
const char *name;
int line_no;
int is_terminate;
child_list_node *child_first_ptr;
};

struct child_list_node {
tree_node *tree_node;
```

```
child_list_node *next_child;
};
...
tree_node* make_tree_node(const char *name, int line_no, int is_terminate) {
    tree_node* node = (tree_node*)malloc(sizeof(tree_node));
    node->name = name;
    node->line_no = line_no ? line_no : yylineno;
    node->child_first_ptr = (child_list_node*)NULL;
    node->is_terminate = is_terminate;
    return node;
}

child_list_node* make_child_list_node(tree_node *tree_node) {
    child_list_node *list_node =
(child_list_node*)malloc(sizeof(child_list_node));
    list_node->tree_node = tree_node;
    list_node->next_child = (child_list_node*)NULL;
    return list_node;
}

void add_child(tree_node *father, tree_node *child) {
    child_list_node *new_child = make_child_list_node(child);
    new_child->next_child = father->child_first_ptr;
    father->child_first_ptr = new_child;
}
```

Error Recovery

Lexical Error

For lexical error, we define some regular expressions to recognize invalid symbols and return a `INVALID` symbol which is a t terminals for syntax analysis.

```
{lexical_err}|. {
    fprintf(stdout, "Error type A at Line %d: unknown lexeme %s\n", yylineno,
yytext);
    return INVALID;
}
```

Syntax Error

For syntax error, we use the error recovery mechanism in Bison: we firstly define the error production expression and then modify the `yyerror()` method.

A example of error recovery in `splc.y`

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
    int has_error = 0;
%}
...
Program:
    ExtDefList {
        $$ = make_tree_node("Program", $1->line_no, 0);
        add_child($$, $1);
```

```
        if (!has_error)
            show_tree($$, 0);
    }
    ;
...
Stmt:
...
| RETURN Exp error {
        printf("Error type B at Line %d: Missing semicolon ';'\n",$1->line_no);
    }
...
void yyerror(const char* msg) {
    has_error=1;
}
```

Conclusion

In this project, we applied the knowledge of theoretical lessons to practice and benefited a lot. Every teammate has made a huge contribution to this project.