Report

12011543 林洁芳 11911212 袁恒宸 12011906 汤奕飞

Basic

Lex

The lex is rather simple, there are three points worth mentioning. First, the regular expression. To support the feature of recognizing A-type error, we must write the regex of possible wrong patterns. When lex read the wrong pattern, it will report an A-type error. Second is about grammar tree. When lex encounters one valid lexeme, it will construct a node containing needed information of that lexeme. The design of node will be discussed later. Third, line number support. We add %option yylineno to support line number.

Bison

Every token and nonterminal in bison is the node* type to construct the tree while parsing. When several tokens and nonterminal are reduced to a new nonterminal, it will form a tree. Bison's error recovery works for recognizing B-type error.

Grammar Tree

node.h and node.c are to define and implement the grammar tree. Void setNode add nodes, and void treePrint with nodePrint print the grammar tree by recursion.

```
4  struct node {
5      int type;
6      int line;
7      char *value;
8      int number;
9      struct node *children[10];
10 };
```

Bonus

Comment

For single-line and multi-line comments, filtering is done mainly by regular expressions. The

code writing part is mainly in the lex.l file.

```
"//".* {}
[/][*][^*]*[*]+([^*/][^*]*[*]+)*[/] {}
```

We designed a test file test_1.spl shown below for annotations, which includes both single-line and multi-line comments. Adding comments at various points in the code, the parse tree can be successfully produced using this test code(in the file named test_1.out).

```
/*
multiline
\/\///\\/
comment
**/
int /*func*/test_1(int m,//)
int n)
{
  int max;
  if (m > /*inline comment*/n)
  {
    max = m;
    // max = 2*m + 1;
  }
  else
  {
    max = n;
  }
  return max;
}
```

Brackets

When we were dealing with incompleteness in “[]” and “{}”, we found that the parse tree was broken because the grammar was not designed in the right order to handle this error. It causes the "Missing Specifier" error to be unchecked. After I processed the incompleteness, using the test_1_r10.spl provided by the teacher as a test example, the following happened.

```
Error type B at Line 4: Missing closing parenthesis '}'
Error type B at Line 5: Missing semicolon ';'
```

```
Error type B at Line 4: Missing specifier
```

Legends

Colors	Links
Added	(f)irst change
Changed	(n)ext change
Deleted	(t)op

float*/ float *& int*/ int *&char*/ char * & void

During parsing, pointers are treated as a data type in lex.l.

```
"void" {yylval = newNodeNL(TYPe,yylineno,yytext); return TYPE;}
"int*"|"int *" {yylval = newNodeNL(TYPe,yylineno,yytext); return TYPE;}
"float*"|"float *" {yylval = newNodeNL(TYPe,yylineno,yytext); return TYPE;}
"char*"|"char *" {yylval = newNodeNL(TYPe,yylineno,yytext); return TYPE;}
```

We still wrote the test code and were able to successfully generate the syntax analysis tree. (In test_2.spl and the output parse tree in test_2.out).

++, --

For symbols, the two are treated as separate terminators in lex and participate in the syntax analysis.
In bison, realize the rationality of their occurrence before or after the ID.

```
void test_3(int len)
{
    int idx = 0;
    int sum = 0;
    char* chs;
    int * ints;
    while (idx < len)
    {
        sum = sum + ints[idx];
        idx++;
    }
    --idx;
}
```

```
/* DOUBLE_MINUS */
| DOUBLE_MINUS Exp
{$$= newNodeNTER(EXp, getLine()); tmpnum = 2;
tmpcld[0] = $1; tmpcld[1] = $2; setNode($$, tmpcld, tmpnum);}
| Exp DOUBLE_MINUS
{$$= newNodeNTER(EXp, getLine()); tmpnum = 2;
tmpcld[0] = $1; tmpcld[1] = $2; setNode($$, tmpcld, tmpnum);}

/* DOUBLE_PLUS */
| DOUBLE_PLUS Exp
{$$= newNodeNTER(EXp, getLine()); tmpnum = 2;
tmpcld[0] = $1; tmpcld[1] = $2; setNode($$, tmpcld, tmpnum);}
| Exp DOUBLE_PLUS
{$$= newNodeNTER(EXp, getLine()); tmpnum = 2;
tmpcld[0] = $1; tmpcld[1] = $2; setNode($$, tmpcld, tmpnum);}
```

The test code and were able to successfully generate the syntax analysis tree. (In test_2.spl and the output parse tree in test_2.out, test with pointer declarations together).

%, +=, -=, *=, /=, %=

Add the resolution of the modulo operator (“%”), “+=”, “-=”, “*=”, “/=”, “%=”, treat the modulo the same as “*” and “/”. In addition, the operators of “+=”, “-=”, “*=”, “/=”, “%=” are similar with “=”.

```
int test_3(int a, int b)
{
    c = 'c';
    // int even = 0;
    if (a > b)
    {
        even = even + 1;
        odd += 1;
        id = id % 2;
        c__ /= 666;
        return a;
    }
    else
    {
        compiler *= 520;
        return b;
    }
}
```

The test code and were able to successfully generate the syntax analysis tree. (In test_3.spl and the output parse tree in test_3.out).

... ? ... : ...

Handle ternary operators and add computational soundness of ternary operators to all statements involving comparison judgments in bison. The test case is test_3.spl and the parse

tree is in test_3.out, testing with the key word “const” as below.

Bison code (a part):

```
| Exp AND Exp QUESTION_MARK Exp COLON Exp
{$$= newNodeNTER(EXp, getLine()); tmpnum = 7; tmpcld[0] = $1;
tmpcld[1] = $2; tmpcld[2] = $3; tmpcld[3] = $4; tmpcld[4] = $5;
tmpcld[5] = $6; tmpcld[6] = $7;
setNode($$, tmpcld, tmpnum);}
```

Const

Supports declaring constants of a specified type using the const prefix.

In lex:

```
"const" {yylval = newNodeTER(CONSt,yylineno); return CONST;}
```

In bison:

```
Specifier:
    TYPE
    {$$= newNodeNTER(SPECIFIEr, getLine()); tmpnum = 1;
    tmpcld[0] = $1; setNode($$, tmpcld, tmpnum);}
    |CONST TYPE
    {$$= newNodeNTER(SPECIFIEr, getLine()); tmpnum = 2;
    tmpcld[0] = $1; tmpcld[1] = $2; setNode($$, tmpcld, tmpnum);}
```

The test code is in test_4.spl and the parse tree is in test_4.out.

```
1  void test_4(int a,int b)
2  {
3      const int max = a >= b ? a : b;
4      const int idx = 0;
5      int sum;
6      while (idx < max)
7      {
8          sum += idx;
9      }
10 }
```