

实验一

词法分析

词法分析的目的是读入外部的字符流（源程序）对其进行扫描，把它们组成有意义的词素序列，对于每个词素，词法分析器都会产生词法单元(Token) 作为输出

语法分析

1. SysY 文法

我们对 SysY 文法进行了一定的限制以减少难度，主要改变是同学们不需要支持二维以上的数组解析、不需要支持各种形式的浮点数字面量解析(不需要支持即我们在测试中不会出现这样的用例)，并对左递归文法做了处理。新的文法请参考 [文法定义](#)。请注意，实现必须以该文法为准

实验一大致分为两个步骤，词法分析：先将文本解析成词法单元序列，语法分析：再将词法单元序列解析成抽象语法树。

数据结构与算法

词法分析相关数据结构

Token(词法单元)

1. Token

Token 的定义在 [token.h](#) 中，同时 Token 类型的枚举类 TokenType 也定义在其中

```
struct Token {
    TokenType type;
    string value;
};

enum class TokenType{
    IDENFR,          // identifier
    INTLTR,           // int literal
    FLOATLTR,        // float literal
    CONSTTK,         // const
    VOIDTK,           // void
    ...
}
```

其中 string value 是 Token 所代表的字符串，TokenType type 是指 Token 的类型

```

1 // enumerate for Token type
2 enum class TokenType
3 {
4     IDENFR,    // identifier
5     INTLTR,    // int literal
6     FLOATLTR,  // float literal
7     CONSTTK,   // const
8     VOIDTK,    // void
9     INTTK,     // int
10    FLOATTK,   // float
11    IFTK,      // if
12    ELSETK,    // else
13    WHILETK,   // while
14    CONTINUEK, // continue
15    BREAKTK,   // break
16    RETURNK,   // return
17    PLUS,      // +
18    MINU,      // -
19    MULT,      // *
20    DIV,       // /
21    MOD,       // %
22    LSS,       // <
23    GTR,       // >
24    COLON,     // :
25    ASSIGN,    // =
26    SEMICN,    // ;
27    COMMA,     // ,
28    LPARENT,   // (
29    RPARENT,   // )
30    LBRACK,    // [
31    RBRACK,    // ]
32    LBRACE,    // {
33    RBRACE,    // }
34    NOT,       // !
35    LEQ,       // <=
36    GEQ,       // >=
37    EQL,       // ==
38    NEQ,       // !=
39    AND,       // &&
40    OR,        // ||
41 };
42 std::string toString(TokenType);
43
44 struct Token
45 {
46     TokenType type;
47     std::string value;
48 };

```

`type` 词法单元的类型

`value` 词法单元的值

DFA(确定性有限自动机)

2. DFA

在词法分析中，我们使用确定有限状态自动机 (deterministic finite automaton, DFA) 来进行分词，对于一个给定的属于该自动机的状态和一个属于该自动机字母表 Σ 的字符，它都能根据事先给定的转移函数转移到下一个状态，某些转移函数会进行输出

我们需要为词法分析设计这样一个 DFA：它可以接收输入字符，进行状态改变，并在某些转移过程中输出累计接受到的字符所组成的字符串

该 DFA 中应存在五种状态，我们用枚举类 State 来表示

```
enum class State {
    Empty,           // space, \n, \r ...
    Ident,           // a keyword or identifier, like 'int' 'a0' 'else' ...
    IntLiteral,      // int literal, like '1' '1900', only in decimal
    FloatLiteral,    // float literal, like '0.1'
    op               // operators and '{', '[', '(', ',', ' ...
};
```

我们将 DFA 及其行为的抽象为类和类方法，定义在 `lexical.h` 中

```

1 struct DFA
2 {
3     /**
4      * @brief constructor, set the init state to State::Empty
5      */
6     DFA();
7
8     /**
9      * @brief destructor
10     */
11     ~DFA();
12
13     // the meaning of copy and assignment for a DFA is not clear, so we do not
14     allow them
15     DFA(const DFA &) = delete; // copy constructor
16     DFA &operator=(const DFA &) = delete; // assignment
17
18     /**
19     * @brief take a char as input, change state to next state, and output a Token
20     if necessary
21     * @param[in] input: the input character
22     * @param[out] buf: the output Token buffer
23     * @return return true if a Token is produced, the buf is valid then
24     */
25     bool next(char input, Token &buf);
26
27     /**
28     * @brief reset the DFA state to begin
29     */
30     void reset();
31
32     /**
33     * @brief get the current state
34     */
35     State get_state() const;
36
37     /**
38     * @brief get the current output Token
39     */
40     Token get_token() const;
41
42     /**
43     * @brief get the current output string
44     */
45     string get_str() const;
46
47     /**
48     * @brief get the current output Token buffer
49     */
50     string get_buf() const;
51
52     /**
53     * @brief get the current output Token buffer
54     */
55     string get_buf() const;
56
57     /**
58     * @brief get the current output Token buffer
59     */
60     string get_buf() const;
61
62     /**
63     * @brief get the current output Token buffer
64     */
65     string get_buf() const;
66
67     /**
68     * @brief get the current output Token buffer
69     */
70     string get_buf() const;
71
72     /**
73     * @brief get the current output Token buffer
74     */
75     string get_buf() const;
76
77     /**
78     * @brief get the current output Token buffer
79     */
80     string get_buf() const;
81
82     /**
83     * @brief get the current output Token buffer
84     */
85     string get_buf() const;
86
87     /**
88     * @brief get the current output Token buffer
89     */
90     string get_buf() const;
91
92     /**
93     * @brief get the current output Token buffer
94     */
95     string get_buf() const;
96
97     /**
98     * @brief get the current output Token buffer
99     */
100    string get_buf() const;
101
102    /**
103     * @brief get the current output Token buffer
104     */
105    string get_buf() const;
106
107    /**
108     * @brief get the current output Token buffer
109     */
110    string get_buf() const;
111
112    /**
113     * @brief get the current output Token buffer
114     */
115    string get_buf() const;
116
117    /**
118     * @brief get the current output Token buffer
119     */
120    string get_buf() const;
121
122    /**
123     * @brief get the current output Token buffer
124     */
125    string get_buf() const;
126
127    /**
128     * @brief get the current output Token buffer
129     */
130    string get_buf() const;
131
132    /**
133     * @brief get the current output Token buffer
134     */
135    string get_buf() const;
136
137    /**
138     * @brief get the current output Token buffer
139     */
140    string get_buf() const;
141
142    /**
143     * @brief get the current output Token buffer
144     */
145    string get_buf() const;
146
147    /**
148     * @brief get the current output Token buffer
149     */
150    string get_buf() const;
151
152    /**
153     * @brief get the current output Token buffer
154     */
155    string get_buf() const;
156
157    /**
158     * @brief get the current output Token buffer
159     */
160    string get_buf() const;
161
162    /**
163     * @brief get the current output Token buffer
164     */
165    string get_buf() const;
166
167    /**
168     * @brief get the current output Token buffer
169     */
170    string get_buf() const;
171
172    /**
173     * @brief get the current output Token buffer
174     */
175    string get_buf() const;
176
177    /**
178     * @brief get the current output Token buffer
179     */
180    string get_buf() const;
181
182    /**
183     * @brief get the current output Token buffer
184     */
185    string get_buf() const;
186
187    /**
188     * @brief get the current output Token buffer
189     */
190    string get_buf() const;
191
192    /**
193     * @brief get the current output Token buffer
194     */
195    string get_buf() const;
196
197    /**
198     * @brief get the current output Token buffer
199     */
200    string get_buf() const;
201
202    /**
203     * @brief get the current output Token buffer
204     */
205    string get_buf() const;
206
207    /**
208     * @brief get the current output Token buffer
209     */
210    string get_buf() const;
211
212    /**
213     * @brief get the current output Token buffer
214     */
215    string get_buf() const;
216
217    /**
218     * @brief get the current output Token buffer
219     */
220    string get_buf() const;
221
222    /**
223     * @brief get the current output Token buffer
224     */
225    string get_buf() const;
226
227    /**
228     * @brief get the current output Token buffer
229     */
230    string get_buf() const;
231
232    /**
233     * @brief get the current output Token buffer
234     */
235    string get_buf() const;
236
237    /**
238     * @brief get the current output Token buffer
239     */
240    string get_buf() const;
241
242    /**
243     * @brief get the current output Token buffer
244     */
245    string get_buf() const;
246
247    /**
248     * @brief get the current output Token buffer
249     */
250    string get_buf() const;
251
252    /**
253     * @brief get the current output Token buffer
254     */
255    string get_buf() const;
256
257    /**
258     * @brief get the current output Token buffer
259     */
260    string get_buf() const;
261
262    /**
263     * @brief get the current output Token buffer
264     */
265    string get_buf() const;
266
267    /**
268     * @brief get the current output Token buffer
269     */
270    string get_buf() const;
271
272    /**
273     * @brief get the current output Token buffer
274     */
275    string get_buf() const;
276
277    /**
278     * @brief get the current output Token buffer
279     */
280    string get_buf() const;
281
282    /**
283     * @brief get the current output Token buffer
284     */
285    string get_buf() const;
286
287    /**
288     * @brief get the current output Token buffer
289     */
290    string get_buf() const;
291
292    /**
293     * @brief get the current output Token buffer
294     */
295    string get_buf() const;
296
297    /**
298     * @brief get the current output Token buffer
299     */
300    string get_buf() const;
301
302    /**
303     * @brief get the current output Token buffer
304     */
305    string get_buf() const;
306
307    /**
308     * @brief get the current output Token buffer
309     */
310    string get_buf() const;
311
312    /**
313     * @brief get the current output Token buffer
314     */
315    string get_buf() const;
316
317    /**
318     * @brief get the current output Token buffer
319     */
320    string get_buf() const;
321
322    /**
323     * @brief get the current output Token buffer
324     */
325    string get_buf() const;
326
327    /**
328     * @brief get the current output Token buffer
329     */
330    string get_buf() const;
331
332    /**
333     * @brief get the current output Token buffer
334     */
335    string get_buf() const;
336
337    /**
338     * @brief get the current output Token buffer
339     */
340    string get_buf() const;
341
342    /**
343     * @brief get the current output Token buffer
344     */
345    string get_buf() const;
346
347    /**
348     * @brief get the current output Token buffer
349     */
350    string get_buf() const;
351
352    /**
353     * @brief get the current output Token buffer
354     */
355    string get_buf() const;
356
357    /**
358     * @brief get the current output Token buffer
359     */
360    string get_buf() const;
361
362    /**
363     * @brief get the current output Token buffer
364     */
365    string get_buf() const;
366
367    /**
368     * @brief get the current output Token buffer
369     */
370    string get_buf() const;
371
372    /**
373     * @brief get the current output Token buffer
374     */
375    string get_buf() const;
376
377    /**
378     * @brief get the current output Token buffer
379     */
380    string get_buf() const;
381
382    /**
383     * @brief get the current output Token buffer
384     */
385    string get_buf() const;
386
387    /**
388     * @brief get the current output Token buffer
389     */
390    string get_buf() const;
391
392    /**
393     * @brief get the current output Token buffer
394     */
395    string get_buf() const;
396
397    /**
398     * @brief get the current output Token buffer
399     */
400    string get_buf() const;
401
402    /**
403     * @brief get the current output Token buffer
404     */
405    string get_buf() const;
406
407    /**
408     * @brief get the current output Token buffer
409     */
410    string get_buf() const;
411
412    /**
413     * @brief get the current output Token buffer
414     */
415    string get_buf() const;
416
417    /**
418     * @brief get the current output Token buffer
419     */
420    string get_buf() const;
421
422    /**
423     * @brief get the current output Token buffer
424     */
425    string get_buf() const;
426
427    /**
428     * @brief get the current output Token buffer
429     */
430    string get_buf() const;
431
432    /**
433     * @brief get the current output Token buffer
434     */
435    string get_buf() const;
436
437    /**
438     * @brief get the current output Token buffer
439     */
440    string get_buf() const;
441
442    /**
443     * @brief get the current output Token buffer
444     */
445    string get_buf() const;
446
447    /**
448     * @brief get the current output Token buffer
449     */
450    string get_buf() const;
451
452    /**
453     * @brief get the current output Token buffer
454     */
455    string get_buf() const;
456
457    /**
458     * @brief get the current output Token buffer
459     */
460    string get_buf() const;
461
462    /**
463     * @brief get the current output Token buffer
464     */
465    string get_buf() const;
466
467    /**
468     * @brief get the current output Token buffer
469     */
470    string get_buf() const;
471
472    /**
473     * @brief get the current output Token buffer
474     */
475    string get_buf() const;
476
477    /**
478     * @brief get the current output Token buffer
479     */
480    string get_buf() const;
481
482    /**
483     * @brief get the current output Token buffer
484     */
485    string get_buf() const;
486
487    /**
488     * @brief get the current output Token buffer
489     */
490    string get_buf() const;
491
492    /**
493     * @brief get the current output Token buffer
494     */
495    string get_buf() const;
496
497    /**
498     * @brief get the current output Token buffer
499     */
500    string get_buf() const;
501
502    /**
503     * @brief get the current output Token buffer
504     */
505    string get_buf() const;
506
```

```

27     */
28     void reset();
29
30 private:
31     State cur_state;    // record current state of the DFA
32     std::string cur_str; // record input characters
33 };

```

`next()` 当DFA.cur_str满足词法单元序列的匹配规则时，返回true,设置buf, 否则为false

`reset()` 重置状态机

Scanner(扫描器)

3. Scanner

Scanner 是扫描器，其职责是将字符串输入转化为 Token 串，词法分析实际上就是实现一个 Scanner

Scanner 的定义在 [lexical.h](#) 中

```

struct Scanner {
    /**
     * @brief constructor
     * @param[in] filename: the input file
     */
    Scanner(std::string filename);

    /**
     * @brief run the scanner, analysis the input file and result a token stream
     * @return std::vector<Token>: the result token stream
     */
    std::vector<Token> run();

private:
    std::ifstream fin; // the input file
};

```

```

1 struct Scanner
2 {
3     /**
4      * @brief constructor
5      * @param[in] filename: the input file
6      */
7     Scanner(std::string filename);
8
9     /**
10     * @brief destructor, close the file
11     */
12     ~Scanner();
13
14     // reject copy and assignment
15     Scanner(const Scanner &) = delete;
16     Scanner &operator=(const Scanner &) = delete;

```

```

17
18     /**
19     * @brief run the scanner, analysis the input file and result a token stream
20     * @return std::vector<Token>: the result token stream
21     */
22     std::vector<Token> run();
23
24 private:
25     std::ifstream fin; // the input file
26 };

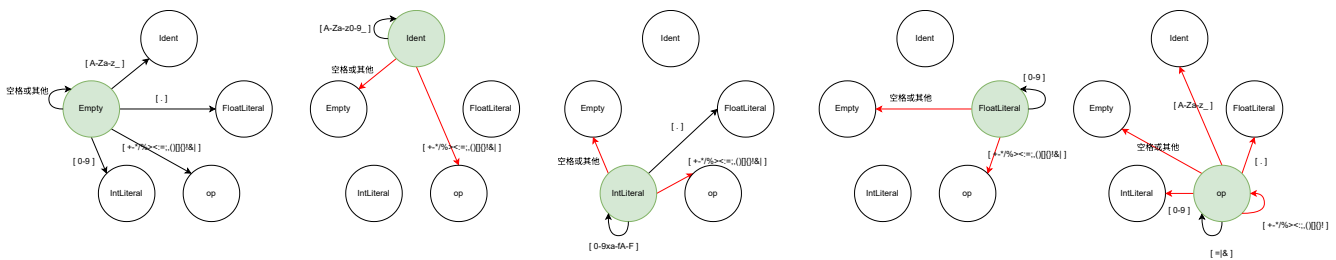
```

`fin` 是C语言程序的文件输入流

`run()` 执行后返回词法单元序列 `std::vector<Token>`

词法分析相关算法

首先来介绍词法分析中最重要的组成部分DFA



DFA类中最重要函数是DFA::next(char input, Token &buf)函数

这个函数的作用是：

1. 当DFA.cur_str满足词法单元序列的匹配规则时，返回true，供scanner类处理;
2. 当next函数返回true时，根据cur_str设置Token&buf的参数，scanner类会将buf添加进 `std::vector<frontend::Token> tk_stream;`中

上图是一个我当时做实验写的一个简单的状态机实例。

比如对于下面这段文本

示例代码

```

1 float a = .2;
2 int b = 0x2;
3 int main(){
4     if(b==2){
5         a = 0.1;
6     }
7 }

```

有如下处理流程

input	cur_str	cur_state	return
初始	none	empty	none
f	f	ident	false
l	fl	ident	false
o	flo	ident	false
a	floa	ident	false
t	float	ident	false
空格	空格	empty	true 处理float
a	a	ident	false
空格	空格	empty	true 处理a
=	=	op	false
.	.	floatliteral	true 处理=
2	.2	floatliteral	false
;	;	op	true 处理.2
int b = 0x2; int main(){ if(b
=	=	op	true 处理b
1	==	op	
2	2	intliteral	true 处理==
{ a = 0.1; } }
空格(最后一次将cur_str清空)	空格	empty	true 处理}

所以得到的输出结果为

token.value	token.type
float	FLOATTK
a	IDENT
=	ASSIGN
.2	FLOATLTR

token.value	token.type
;	SEMICN
int	INTTK
b	IDENT
=	ASSIGN
0x2	INTLTR
int	INTTK
main	IDENT
(LPARENT
)	RPARENT
{	LBRACE
if	RBRACK
(LPARENT
b	IDENT
==	EQL
2	INTLTR
)	RPARENT
{	LBRACE
a	IDENT
=	ASSIGN
0.1	FLOATLTR
;	SEMICN
}	RBRACE
}	RBRACE

补充说明

1. 这里没有考虑注释的情况，如果出现 `//注释` 和 `/*多行注释*/` 的情况下怎么办？
2. 有限自动机也是正则表达式算法的核心实现，这里的程序是否能使用[正则表达式](#)来重构？

语法分析相关数据结构

AST(抽象语法树)

```
1  enum class NodeType {
2      TERMINAL, // terminal lexical unit
3      COMPUINT,
4      DECL,
5      FUNCDEF,
6      CONSTDECL,
7      BTYPE,
8      CONSTDEF,
9      CONSTINITVAL,
10     VARDECL,
11     VARDEF,
12     INITVAL,
13     FUNCTYPE,
14     FUNCFPARAM,
15     FUNCFPARAMS,
16     BLOCK,
17     BLOCKITEM,
18     STMT,
19     EXP,
20     COND,
21     LVAL,
22     NUMBER,
23     PRIMARYEXP,
24     UNARYEXP,
25     UNARYOP,
26     FUNCRPARAMS,
27     MULEXP,
28     ADDEXP,
29     RELEXP,
30     EQEXP,
31     LANDEXP,
32     LOREXP,
33     CONSTEXP,
34 };
35 std::string toString(NodeType);
36
37 struct Variant
38 {
39     Type t;
40     Operand v_int;
41     Operand v_float;
42 };
43
44 // tree node basic class
45 struct AstNode
46 {
47     NodeType type; // the node type
48     AstNode* parent; // the parent node
```



```

49     vector<AstNode*> children; // children of node
50
51     /**
52      * @brief constructor
53      */
54     AstNode(NodeType t, AstNode* p = nullptr);
55
56     /**
57      * @brief destructor
58      */
59     virtual ~AstNode();
60
61     /**
62      * @brief Get the json output object
63      * @param root: a Json::Value buffer, should be initialized before calling
64      * this function
65      */
66     void get_json_output(Json::Value& root) const;
67
68     // reject copy and assignment
69     AstNode(const AstNode&) = delete;
70     AstNode& operator=(const AstNode&) = delete;
71 };

```

`type` 抽象语法树节点类型

`parent` 父母节点

`children` 子节点

`get_json_output` 获取json形式的抽象语法树输出

Parser(语法分析器)

```

1  struct Parser
2  {
3      uint32_t          index; // current token index
4      const std::vector<Token>& token_stream;
5
6      /**
7       * @brief constructor
8       * @param tokens: the input token_stream
9       */
10     Parser(const std::vector<Token>& tokens);
11
12     /**
13      * @brief destructor
14      */
15     ~Parser();
16
17     /**
18      * @brief creat the abstract syntax tree
19      * @return the root of abstract syntax tree

```

```

20     */
21     CompUnit* get_abstract_syntax_tree();
22     /**
23      * @brief for debug, should be called in the beginning of recursive descent
24      * functions
25      * @param node: current parsing node
26      */
27     void log(AstNode* node);
28
29     bool parseTerm(AstNode* root, TokenType expect);
30     bool parseCompUnit(AstNode* root);
31     bool parseDecl(AstNode* root);
32     bool parseFuncDef(AstNode* root);
33     bool parseConstDecl(AstNode* root);
34     bool parseBType(AstNode* root);
35     bool parseConstDef(AstNode* root);
36     bool parseConstInitVal(AstNode* root);
37     bool parseVarDecl(AstNode* root);
38     bool parseVarDef(AstNode* root);
39     bool parseInitVal(AstNode* root);
40     bool parseFuncType(AstNode* root);
41     bool parseFuncFParam(AstNode* root);
42     bool parseFuncFParams(AstNode* root);
43     bool parseBlock(AstNode* root);
44     bool parseBlockItem(AstNode* root);
45     bool parseStmt(AstNode* root);
46     bool parseExp(AstNode* root);
47     bool parseCond(AstNode* root);
48     bool parseLVal(AstNode* root);
49     bool parseNumber(AstNode* root);
50     bool parsePrimaryExp(AstNode* root);
51     bool parseUnaryExp(AstNode* root);
52     bool parseUnaryOp(AstNode* root);
53     bool parseFuncRParams(AstNode* root);
54     bool parseMulExp(AstNode* root);
55     bool parseAddExp(AstNode* root);
56     bool parseRelExp(AstNode* root);
57     bool parseEqExp(AstNode* root);
58     bool parseLAndExp(AstNode* root);
59     bool parseLOrExp(AstNode* root);
60     bool parseConstExp(AstNode* root);
61 };

```

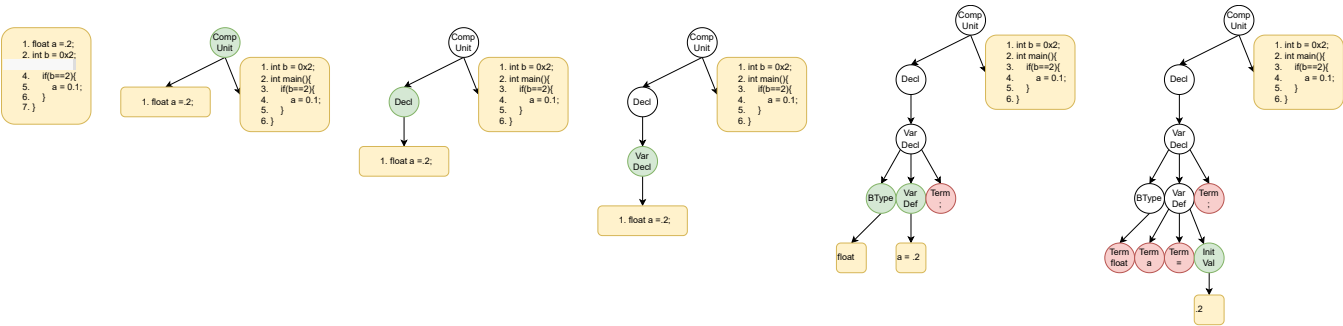
`index` 下一个解析的词法单元

`token_stream` 词法单元序列

`get_abstract_syntax_tree()` 抽象语法树接口

`bool parseXXX(AstNode* root)` 解析特定的抽象语法树节点，解析成功则返回true

语法分析相关算法



- 符号 [...] 表示方括号内包含的为可选项；
- 符号 { ... } 表示花括号内包含的为可重复 0 次或多次的项；
- 终结符或者是单引号括起的串，或者是 Ident、InstConst、floatConst 这样的记号

产生式
CompUnit → (Decl FuncDef) [CompUnit]
Decl → ConstDecl VarDecl
ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
BType → 'int' 'float'
ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
ConstInitVal → ConstExp '{' [ConstInitVal { ',' ConstInitVal }] '}'
VarDecl → BType VarDef { ',' VarDef } ';'
VarDef → Ident { '[' ConstExp ']' } ['=' InitVal]
InitVal → Exp '{' [InitVal { ',' InitVal }] '}'
FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
FuncType → 'void' 'int' 'float'
FuncFParam → BType Ident '[' ']' { '[' Exp ']' }
FuncFParams → FuncFParam { ',' FuncFParam }
Block → '{' { BlockItem } '}'
BlockItem → Decl Stmt
Stmt → LVal '=' Exp ';' Block 'if' '(' Cond ')' Stmt ['else' Stmt] 'while' '(' Cond ')' Stmt 'break' ';' 'continue' ';' 'return' [Exp] ';' [Exp] ';'
Exp → AddExp

产生式
$\text{Cond} \rightarrow \text{LOrExp}$
$\text{LVal} \rightarrow \text{Ident} \{ '[\text{Exp}]' \}$
$\text{Number} \rightarrow \text{IntConst} \mid \text{floatConst}$
$\text{PrimaryExp} \rightarrow '(' \text{Exp} ')' \mid \text{LVal} \mid \text{Number}$
$\text{UnaryExp} \rightarrow \text{PrimaryExp} \mid \text{Ident} '(' [\text{FuncRParams}] ')' \mid \text{UnaryOp} \text{UnaryExp}$
$\text{UnaryOp} \rightarrow '+' \mid '-' \mid '!'$
$\text{FuncRParams} \rightarrow \text{Exp} \{ ',' \text{Exp} \}$
$\text{MulExp} \rightarrow \text{UnaryExp} \{ ('*' \mid '/' \mid '\%') \text{UnaryExp} \}$
$\text{AddExp} \rightarrow \text{MulExp} \{ ('+' \mid '-') \text{MulExp} \}$
$\text{RelExp} \rightarrow \text{AddExp} \{ ('<' \mid '>' \mid '<=' \mid '>=') \text{AddExp} \}$
$\text{EqExp} \rightarrow \text{RelExp} \{ ('==' \mid '!=') \text{RelExp} \}$
$\text{LAndExp} \rightarrow \text{EqExp} ['\&\&' \text{LAndExp}]$
$\text{LOrExp} \rightarrow \text{LAndExp} [' ' \text{LOrExp}]$
$\text{ConstExp} \rightarrow \text{AddExp}$

！要通过实验一的检测程序**必须**按照上述产生式进行抽象语法树构建

最终我们要实现syntax.cpp文件下 `get_abstract_syntax_tree()` 抽象语法树接口。

★这部分的代码主要用到了回溯和递归的思想

`parsexxx()` 函数主要做了如下两件事：

1. 当解析XXX类型节点成功时，返回true
2. 设置index在正确的位置(解析成功时为下一个词法单元索引，失败时则为解析前的词法单元索引)

示例1:产生式 $A \rightarrow BC$

```
1  Function ParseA(ASTNode* root)
2  Begin
3      保存index数据
4      b = ParseB(第一个子节点)
5      如果b为假，恢复index，返回false
6      //b为真
7      c = ParseC(第一个子节点)
8      如果c为假，恢复index，返回false
9      创建A节点，将节点的父节点设为root，将自己加入root的子节点
10     返回true
11 End
```

示例2:产生式 $A \rightarrow B|C$

```
1  Function ParseA(ASTNode* root)
2  Begin
3      保存index数据
4      b = ParseB(第一个子节点)
5      如果b为真, 创建A节点, 将节点的父节点设为root, 将自己加入root的子节点, 返回true
6      //b为假
7      c = ParseC(第一个子节点)
8      如果c为真, 创建A节点, 将节点的父节点设为root, 将自己加入root的子节点, 返回true
9      恢复index
10     返回false
11 End
```

示例3:产生式 $A \rightarrow B B$ 为终结符

```
1  Function ParseA(ASTNode* root)
2  Begin
3      保存index数据
4      如果index对应词法单元和B不一致, 恢复index, 返回false
5      //尝试成功
6      index++
7      创建Term点, 将节点的父节点设为root, 将自己加入root的子节点
8
9      返回true
10 End
```

★ 示例4:优先级问题

对于如下一段特殊代码

```
1  f();
```

假设目前我们已经分析到了UnaryExp节点, 对于上文定义文法如下

$\text{UnaryExp} \rightarrow \text{PrimaryExp} \mid \text{Ident '(' [FuncRParams] ')'} \mid \text{UnaryOp UnaryExp}$

这里应该采用第二条 `Ident '(' [FuncRParams] ')'` 产生式。但是如果按照先后顺序分析, 我们会先尝试分析 `PrimaryExp`, 那么它的分析路径如下:

`PrimaryExp` \rightarrow `Lval` \rightarrow `'f'`

分析成功, 返回true, 然后 `ParseUnaryExp` 也返回true, 此时index指向的是 `(`, 然后程序递归到 `ParseMulExp`, `ParseMulExp` 尝试分析 `* / %` 失败,直到所有程序都尝试解析 `(` 失败, 程序报错。

解决方式：将第二条 Ident '(' [FuncRParams] ')' 产生式提前分析。先分析难分析的，再分析简单的，采用贪婪模式。

★ 示例5:另一种接口

上面提到的示例接口是

```
1 | bool parseXXXXX(AstNode* root);
```

需要将父节点传入，供子节点使用，其实有另一种更加简便的定义接口的办法。

```
1 | AstNode* parseXXXXX();
```

这里不需要传入父节点，而是将子节点返回给父节点，让父节点成为构建树的负责人，当构建成功时返回子节点，构建失败返回nullptr。

那么对于产生式 $A \rightarrow BC$ ，有伪代码

```
1 | Function ParseA(ASTNode* root)
2 | Begin
3 |     保存index数据
4 |     b = ParseB(第一个子节点)
5 |     如果b为nullptr, 恢复index, 返回false
6 |     //b为真
7 |     c = ParseC(第一个子节点)
8 |     如果c为nullptr, 恢复index, 返回false
9 |     创建a = A节点, 将b,c加入自己的子节点
10 |    返回a
11 | End
```

Q&A