2015-11-18

编译原理实验

语法分析器的简单实现

南京大学软件学院 王振聪 131250218

1. 实验目的

通过构建一个简单的语法分析程序，进一步熟悉语法分析的过程以及加深对 First 和 Follow 的认识。

2. 实验描述

本次实验使用 Java 语言编写，简单实现了对 C 语言程序的语法分析。程序的输入是 C 语言程序代码 program.c 文件以及已经定义好的文法 CFG.txt 文件。程序的输出是控制台和 output.txt 文件，内容是语法分析的过程。本实验中用到的词法分析部分是实验一的词法分析器。

3. 实验方法

本次实验选用的方法是 PPT 中给出的第一种方法中的 a 方法，根据文法编程构建 LL(1) 预测分析表来进行语法分析。

4. 实验设想

暂时不处理二义文法。

5. 相关的自动机描述

大部分同词法分析。

6. 数据结构的定义

Lex 类：词法分析程序
Token 类：Token 序列中的元素
Production 类：产生式类，包括两个属性，一个是 left（String 类型），表示产生式的左部；另一个是 right（List<String>类型），表示产生式的右部。
FirstFollow 类：表示 First 或 Follow 的类，包括两个属性，一个是 left（String 类型），另一个是 right（List<String>类型）。
Syntax 类：语法分析程序核心

终结符和非终结符分别用 List<String>类型来表示。

产生式用 List<Production>类型来表示。

符号的 First 和 Follow 分别用 List<FirstFollow>类型来表示。

预测分析表用 Production[][]类型表示，还需要借助终结符和非终结符的 List<String>作为表头。

语法分析过程中的栈用 Java 自带的 Stack<String>来表示。

Token 序列（简化版）用 List<String>类型来表示。

7. 核心算法描述

程序的核心算法主要有四个：

1. getFirst()：获得符号的 First 集合，算法在龙书的 P140.
2. getFollow()：获得符号的 Follow 集合，算法在龙书的 P140-141.
3. getParsingTable()：构建预测分析表，算法在龙书的 P142-143，算法 4.31
4. syntaxParsing()：表驱动的预测语法分析，算法在龙书的 P144，算法 4.34

8. 测试用例

C 语言程序源代码：(program.c)

```
int main(float a) {
	while (a <= 5) {
		if (a == 6) {
			a = 8;
		} else {
			int b = 5;
			b = a + 5;
		}
	}
	return 0;
}
```

文法：(CFG.txt)

```
<Program> -> <Function>
<Function> -> [void] <FuncName> [(] <Para> [)] [{] <Statement> [}]
<Function> -> <DataType> <FuncName> [(] <Para> [)] [{] <Statement> <Return> [}]
<FuncName> -> [ID]
<Para> -> [ε]
<Para> -> <DataType> [ID]
<Statement> -> <S> <Statement>
<Statement> -> [ε]
<S> -> <DataType> [ID] <M>
<M> -> [;]
<M> -> [=] [NUM] [;]
<S> -> [ID] [=] <Keyword> <N>
<N> -> [;]
<N> -> <OP> <Keyword> [;]
<S> -> [if] [(] <C> [)] <B> <D>
<D> -> [else] <B>
<D> -> [ε]
<B> -> [{] <Statement> [}]
<C> -> <Keyword> <CompareOP> <Keyword>
<S> -> [while] [(] <C> [)] <B>
<DataType> -> [int]
<DataType> -> [float]
```

```
<DataType> -> [double]
<OP> -> [+]
<OP> -> [-]
<OP> -> [*]
<OP> -> [/]
<OP> -> [%]
<CompareOP> -> [==]
<CompareOP> -> [!=]
<CompareOP> -> [>=]
<CompareOP> -> [<=]
<CompareOP> -> [>]
<CompareOP> -> [<]
<Keyword> -> [ID]
<Keyword> -> [NUM]
<Return> -> [return] <Keyword> [;]
```

下面是一些可选择的输出：(Syntax 类中的 public 方法)：

outputTerminals()：

```
void ( ) { } ID ε ; = NUM if else while int float double + - * / % == != >= <= > < return $
```

outputNonTerminals()：

```
Program Function FuncName Para Statement DataType Return S M Keyword N OP C B D CompareOP
```

outputProductions()：

```
Program -> Function
Function -> void FuncName ( Para ) { Statement }
Function -> DataType FuncName ( Para ) { Statement Return }
FuncName -> ID
Para -> ε
Para -> DataType ID
Statement -> S Statement
Statement -> ε
S -> DataType ID M
M -> ;
M -> = NUM ;
S -> ID = Keyword N
N -> ;
N -> OP Keyword ;
S -> if ( C ) B D
D -> else B
D -> ε
```

```
B -> { Statement }
C -> Keyword CompareOP Keyword
S -> while ( C ) B
DataType -> int
DataType -> float
DataType -> double
OP -> +
OP -> -
OP -> *
OP -> /
OP -> %
CompareOP -> ==
CompareOP -> !=
CompareOP -> >=
CompareOP -> <=
CompareOP -> >
CompareOP -> <
Keyword -> ID
Keyword -> NUM
Return -> return Keyword ;
```

outputFirstFollows() :

```
First( void ) → void
First( ( ) → (
First( ) ) → )
First( { ) → {
First( } ) → }
First( ID ) → ID
First( ε ) → ε
First( ; ) → ;
First( = ) → =
First( NUM ) → NUM
First( if ) → if
First( else ) → else
First( while ) → while
First( int ) → int
First( float ) → float
First( double ) → double
First( + ) → +
First( - ) → -
First( * ) → *
First( / ) → /
First( % ) → %
First( == ) → ==
```

First(!=) → !=
First(>=) → >=
First(<=) → <=
First(>) → >
First(<) → <
First(return) → return
First($) → $
First(Program) → void int float double
First(Function) → void int float double
First(FuncName) → ID
First(Para) → ε int float double
First(Statement) → ε int float double ID if while
First(DataType) → int float double
First(Return) → return
First(S) → int float double ID if while
First(M) → ; =
First(Keyword) → ID NUM
First(N) → ; + - * / %
First(OP) → + - * / %
First(C) → ID NUM
First(B) → {
First(D) → ε else
First(CompareOP) → == != >= <= > <
Follow(void) → ID
Follow(( ) → int float double ID NUM
Follow( )) → {
Follow({) → int float double ID if while
Follow(}) → $ else int float double ID if while } return
Follow(ID) → ; = ( ) + - * / % == != >= <= > <
Follow(ε) →) } return int float double ID if while
Follow(;) → } int float double ID if while return
Follow(=) → ID NUM
Follow(NUM) → ; + - * / % == != >= <= > <)
Follow(if) → (
Follow(else) → {
Follow(while) → (
Follow(int) → ID
Follow(float) → ID
Follow(double) → ID
Follow(+) → ID NUM
Follow(-) → ID NUM
Follow(*) → ID NUM
Follow(/) → ID NUM
Follow(%) → ID NUM

```
Follow( == ) → ID NUM
Follow( != ) → ID NUM
Follow( >= ) → ID NUM
Follow( <= ) → ID NUM
Follow( > ) → ID NUM
Follow( < ) → ID NUM
Follow( return ) → ID NUM
Follow( $ ) →
Follow( Program ) → $
Follow( Function ) → $
Follow( FuncName ) → (
Follow( Para ) → )
Follow( Statement ) → } return
Follow( DataType ) → ID
Follow( Return ) → }
Follow( S ) → int float double ID if while } return
Follow( M ) → int float double ID if while } return
Follow( Keyword ) → ; + - * / % == != >= <= > < )
Follow( N ) → int float double ID if while } return
Follow( OP ) → ID NUM
Follow( C ) → )
Follow( B ) → else int float double ID if while } return
Follow( D ) → int float double ID if while } return
Follow( CompareOP ) → ID NUM
```

outputPPT()：

```
<Program> [void] Program -> Function
<Program> [(]    _
<Program> [)]    _
<Program> [{]    _
<Program> [}]    _
<Program> [ID]   _
<Program> [ε]    _
<Program> [;]    _
<Program> [=]    _
<Program> [NUM]     _
<Program> [if]   _
<Program> [else] _
<Program> [while]     _
<Program> [int]   Program -> Function
<Program> [float] Program -> Function
<Program> [double]   Program -> Function
<Program> [+]    _
<Program> [-]    _
```

<Program> [*] _
<Program> [/] _
<Program> [%] _
<Program> [==] _
<Program> [!=] _
<Program> [>=] _
<Program> [<=] _
<Program> [>] _
<Program> [<] _
<Program> [return] _
<Program> [$] _
<Function> [void] Function -> void FuncName (Para) { Statement }
<Function> [(] _
<Function> [)] _
<Function> [{] _
<Function> [}] _
<Function> [ID] _
<Function> [ε] _
<Function> [;] _
<Function> [=] _
<Function> [NUM] _
<Function> [if] _
<Function> [else] _
<Function> [while] _
<Function> [int] Function -> DataType FuncName (Para) { Statement Return }
<Function> [float] Function -> DataType FuncName (Para) { Statement Return }
<Function> [double] Function -> DataType FuncName (Para) { Statement Return }
<Function> [+] _
<Function> [-] _
<Function> [*] _
<Function> [/] _
<Function> [%] _
<Function> [==] _
<Function> [!=] _
<Function> [>=] _
<Function> [<=] _
<Function> [>] _
<Function> [<] _
<Function> [return] _
<Function> [$] _
<FuncName> [void] _
<FuncName> [(] _
<FuncName> [)] _
<FuncName> [{] _

<FuncName> [}] _
<FuncName> [ID] FuncName -> ID
<FuncName> [ε] _
<FuncName> [;] _
<FuncName> [=] _
<FuncName> [NUM] _
<FuncName> [if] _
<FuncName> [else] _
<FuncName> [while] _
<FuncName> [int]_
<FuncName> [float] _
<FuncName> [double]_
<FuncName> [+] _
<FuncName> [-] _
<FuncName> [*] _
<FuncName> [/] _
<FuncName> [%] _
<FuncName> [==] _
<FuncName> [!=]_
<FuncName> [>=] _
<FuncName> [<=] _
<FuncName> [>] _
<FuncName> [<] _
<FuncName> [return] _
<FuncName> [$] _
<Para> [void]_
<Para> [(] _
<Para> [)] Para -> ε
<Para> [{] _
<Para> [}] _
<Para> [ID] _
<Para> [ε] Para -> ε
<Para> [;] _
<Para> [=] _
<Para> [NUM] _
<Para> [if] _
<Para> [else]_
<Para> [while] _
<Para> [int] Para -> DataType ID
<Para> [float]Para -> DataType ID
<Para> [double] Para -> DataType ID
<Para> [+] _
<Para> [-] _
<Para> [*] _

<Para> [/] _
<Para> [%] _
<Para> [==] _
<Para> [!=] _
<Para> [>=] _
<Para> [<=] _
<Para> [>] _
<Para> [<] _
<Para> [return] _
<Para> [$] _
<Statement> [void] _
<Statement> [(] _
<Statement> [)] _
<Statement> [{] _
<Statement> [}] Statement -> ε
<Statement> [ID] Statement -> S Statement
<Statement> [ε] Statement -> ε
<Statement> [;] _
<Statement> [=] _
<Statement> [NUM] _
<Statement> [if] Statement -> S Statement
<Statement> [else] _
<Statement> [while] Statement -> S Statement
<Statement> [int] Statement -> S Statement
<Statement> [float] Statement -> S Statement
<Statement> [double] Statement -> S Statement
<Statement> [+] _
<Statement> [-] _
<Statement> [*] _
<Statement> [/] _
<Statement> [%] _
<Statement> [==]_
<Statement> [!=] _
<Statement> [>=]_
<Statement> [<=]_
<Statement> [>] _
<Statement> [<] _
<Statement> [return] Statement -> ε
<Statement> [$] _
<DataType> [void] _
<DataType> [(] _
<DataType> [)] _
<DataType> [{] _
<DataType> [}] _

<DataType> [ID] _
<DataType> [ε] _
<DataType> [;] _
<DataType> [=] _
<DataType> [NUM] _
<DataType> [if] _
<DataType> [else]_
<DataType> [while] _
<DataType> [int] DataType -> int
<DataType> [float] DataType -> float
<DataType> [double] DataType -> double
<DataType> [+] _
<DataType> [-] _
<DataType> [*] _
<DataType> [/] _
<DataType> [%] _
<DataType> [==] _
<DataType> [!=] _
<DataType> [>=] _
<DataType> [<=] _
<DataType> [>] _
<DataType> [<] _
<DataType> [return] _
<DataType> [$] _
<Return> [void] _
<Return> [(] _
<Return> [)] _
<Return> [{] _
<Return> [}] _
<Return> [ID]_
<Return> [ε] _
<Return> [;] _
<Return> [=] _
<Return> [NUM] _
<Return> [if] _
<Return> [else] _
<Return> [while] _
<Return> [int] _
<Return> [float] _
<Return> [double] _
<Return> [+] _
<Return> [-] _
<Return> [*] _
<Return> [/] _

<Return> [%] _
<Return> [==] _
<Return> [!=]_
<Return> [>=] _
<Return> [<=] _
<Return> [>] _
<Return> [<] _
<Return> [return] Return -> return Keyword ;
<Return> [$] _
<S> [void] _
<S> [(] _
<S> [)] _
<S> [{] _
<S> [}] _
<S> [ID] S -> ID = Keyword N
<S> [ε] _
<S> [;] _
<S> [=] _
<S> [NUM] _
<S> [if] S -> if (C) B D
<S> [else] _
<S> [while] S -> while (C) B
<S> [int] S -> DataType ID M
<S> [float] S -> DataType ID M
<S> [double] S -> DataType ID M
<S> [+] _
<S> [-] _
<S> [*] _
<S> [/] _
<S> [%] _
<S> [==] _
<S> [!=] _
<S> [>=] _
<S> [<=] _
<S> [>] _
<S> [<] _
<S> [return] _
<S> [$] _
<M> [void] _
<M> [(] _
<M> [)] _
<M> [{] _
<M> [}] _
<M> [ID]_

<M> [ε] _
<M> [;] M -> ;
<M> [=] M -> = NUM ;
<M> [NUM] _
<M> [if] _
<M> [else] _
<M> [while] _
<M> [int] _
<M> [float] _
<M> [double] _
<M> [+] _
<M> [-] _
<M> [*] _
<M> [/] _
<M> [%] _
<M> [==] _
<M> [!=] _
<M> [>=] _
<M> [<=] _
<M> [>] _
<M> [<] _
<M> [return] _
<M> [$] _
<Keyword> [void] _
<Keyword> [(] _
<Keyword> [)] _
<Keyword> [{] _
<Keyword> [}] _
<Keyword> [ID] Keyword -> ID
<Keyword> [ε] _
<Keyword> [;] _
<Keyword> [=] _
<Keyword> [NUM] Keyword -> NUM
<Keyword> [if] _
<Keyword> [else] _
<Keyword> [while] _
<Keyword> [int] _
<Keyword> [float] _
<Keyword> [double] _
<Keyword> [+] _
<Keyword> [-] _
<Keyword> [*] _
<Keyword> [/] _
<Keyword> [%] _

<Keyword> [==] _
<Keyword> [!=] _
<Keyword> [>=] _
<Keyword> [<=] _
<Keyword> [>] _
<Keyword> [<] _
<Keyword> [return] _
<Keyword> [$] _
<N> [void] _
<N> [(] _
<N> [)] _
<N> [{] _
<N> [}] _
<N> [ID] _
<N> [ε] _
<N> [;] N -> ;
<N> [=] _
<N> [NUM] _
<N> [if] _
<N> [else] _
<N> [while] _
<N> [int] _
<N> [float] _
<N> [double] _
<N> [+] N -> OP Keyword ;
<N> [-] N -> OP Keyword ;
<N> [*] N -> OP Keyword ;
<N> [/] N -> OP Keyword ;
<N> [%] N -> OP Keyword ;
<N> [==] _
<N> [!=] _
<N> [>=] _
<N> [<=] _
<N> [>] _
<N> [<] _
<N> [return] _
<N> [$] _
<OP> [void] _
<OP> [(] _
<OP> [)] _
<OP> [{] _
<OP> [}] _
<OP> [ID] _
<OP> [ε] _

<OP> [;] _
<OP> [=] _
<OP> [NUM] _
<OP> [if] _
<OP> [else] _
<OP> [while] _
<OP> [int] _
<OP> [float] _
<OP> [double] _
<OP> [+] OP -> +
<OP> [-] OP -> -
<OP> [*]OP -> *
<OP> [/]OP -> /
<OP> [%] OP -> %
<OP> [==] _
<OP> [!=] _
<OP> [>=] _
<OP> [<=] _
<OP> [>] _
<OP> [<] _
<OP> [return] _
<OP> [$] _
<C> [void] _
<C> [(] _
<C> [)] _
<C> [{] _
<C> [}] _
<C> [ID] C -> Keyword CompareOP Keyword
<C> [ε] _
<C> [;] _
<C> [=] _
<C> [NUM] C -> Keyword CompareOP Keyword
<C> [if] _
<C> [else] _
<C> [while] _
<C> [int]_
<C> [float] _
<C> [double]_
<C> [+] _
<C> [-] _
<C> [*] _
<C> [/] _
<C> [%] _
<C> [==] _

<C> [!=] _
<C> [>=] _
<C> [<=] _
<C> [>] _
<C> [<] _
<C> [return] _
<C> [$] _
<B> [void] _
<B> [(] _
<B> [)] _
<B> [{] B -> { Statement }
<B> [}] _
<B> [ID] _
<B> [ε] _
<B> [;] _
<B> [=] _
<B> [NUM] _
<B> [if] _
<B> [else] _
<B> [while] _
<B> [int] _
<B> [float] _
<B> [double] _
<B> [+] _
<B> [-] _
<B> [*] _
<B> [/] _
<B> [%] _
<B> [==] _
<B> [!=] _
<B> [>=] _
<B> [<=] _
<B> [>] _
<B> [<] _
<B> [return] _
<B> [$] _
<D> [void] _
<D> [(] _
<D> [)] _
<D> [{] _
<D> [}] D -> ε
<D> [ID] D -> ε
<D> [ε] D -> ε
<D> [;] _

<D> [=] _
<D> [NUM] _
<D> [if] D -> ε
<D> [else] D -> else B
<D> [while] D -> ε
<D> [int]D -> ε
<D> [float] D -> ε
<D> [double] D -> ε
<D> [+] _
<D> [-] _
<D> [*] _
<D> [/] _
<D> [%] _
<D> [==] _
<D> [!=] _
<D> [>=] _
<D> [<=] _
<D> [>] _
<D> [<] _
<D> [return] D -> ε
<D> [$] _
<CompareOP> [void] _
<CompareOP> [(] _
<CompareOP> [)] _
<CompareOP> [{] _
<CompareOP> [}] _
<CompareOP> [ID] _
<CompareOP> [ε] _
<CompareOP> [;] _
<CompareOP> [=] _
<CompareOP> [NUM] _
<CompareOP> [if] _
<CompareOP> [else] _
<CompareOP> [while] _
<CompareOP> [int] _
<CompareOP> [float] _
<CompareOP> [double] _
<CompareOP> [+] _
<CompareOP> [-] _
<CompareOP> [*] _
<CompareOP> [/] _
<CompareOP> [%] _
<CompareOP> [==] CompareOP -> ==
<CompareOP> [!=] CompareOP -> !=

```
<CompareOP> [>=]    CompareOP -> >=
<CompareOP> [<=]    CompareOP -> <=
<CompareOP> [>]     CompareOP -> >
<CompareOP> [<]     CompareOP -> <
<CompareOP> [return]      _
<CompareOP> [$]_
```

语法分析的过程输出 output.txt：

```
Match:
Stack:          $ Function
Input:          int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:         输出: Program -> Function

Match:
Stack:          $ } Return Statement { ) Para ( FuncName DataType
Input:          int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:         输出: Function -> DataType FuncName ( Para ) { Statement Return }

Match:
Stack:          $ } Return Statement { ) Para ( FuncName int
Input:          int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:         输出: DataType -> int

Match:          int
Stack:          $ } Return Statement { ) Para ( FuncName
Input:          ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else { int
ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:         匹配 int

Match:          int
Stack:          $ } Return Statement { ) Para ( ID
Input:          ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else { int
ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:         输出: FuncName -> ID

Match:          int ID
Stack:          $ } Return Statement { ) Para (
Input:          ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else { int ID
= NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:         匹配 ID
```

Match: int ID (
Stack: $ } Return Statement {) Para
Input: float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 (

Match: int ID (
Stack: $ } Return Statement {) ID DataType
Input: float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: Para -> DataType ID

Match: int ID (
Stack: $ } Return Statement {) ID float
Input: float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: DataType -> float

Match: int ID (float
Stack: $ } Return Statement {) ID
Input: ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 float

Match: int ID (float ID
Stack: $ } Return Statement {)
Input:) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 ID

Match: int ID (float ID)
Stack: $ } Return Statement {
Input: { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配)

Match: int ID (float ID) {
Stack: $ } Return Statement
Input: while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 {

Match: int ID (float ID) {
Stack: $ } Return Statement S

Input: while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: Statement -> S Statement

Match: int ID (float ID) {
Stack: $ } Return Statement B) C (while
Input: while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: S -> while (C) B

Match: int ID (float ID) { while
Stack: $ } Return Statement B) C (
Input: (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 while

Match: int ID (float ID) { while (
Stack: $ } Return Statement B) C
Input: ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 (

Match: int ID (float ID) { while (
Stack: $ } Return Statement B) Keyword CompareOP Keyword
Input: ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: C -> Keyword CompareOP Keyword

Match: int ID (float ID) { while (
Stack: $ } Return Statement B) Keyword CompareOP ID
Input: ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: Keyword -> ID

Match: int ID (float ID) { while (ID
Stack: $ } Return Statement B) Keyword CompareOP
Input: <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 ID

Match: int ID (float ID) { while (ID
Stack: $ } Return Statement B) Keyword <=
Input: <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $

Action: 输出: CompareOP -> <=

Match: int ID (float ID) { while (ID <=
Stack: $ } Return Statement B) Keyword
Input: NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } }
return NUM ; } $
Action: 匹配 <=

Match: int ID (float ID) { while (ID <=
Stack: $ } Return Statement B) NUM
Input: NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } }
return NUM ; } $
Action: 输出: Keyword -> NUM

Match: int ID (float ID) { while (ID <= NUM
Stack: $ } Return Statement B)
Input:) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } }
return NUM ; } $
Action: 匹配 NUM

Match: int ID (float ID) { while (ID <= NUM)
Stack: $ } Return Statement B
Input: { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return
NUM ; } $
Action: 匹配)

Match: int ID (float ID) { while (ID <= NUM)
Stack: $ } Return Statement } Statement {
Input: { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return
NUM ; } $
Action: 输出: B -> { Statement }

Match: int ID (float ID) { while (ID <= NUM) {
Stack: $ } Return Statement } Statement
Input: if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return
NUM ; } $
Action: 匹配 {

Match: int ID (float ID) { while (ID <= NUM) {
Stack: $ } Return Statement } Statement S
Input: if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return
NUM ; } $
Action: 输出: Statement -> S Statement

Match: int ID (float ID) { while (ID <= NUM) {
Stack: $ } Return Statement } Statement D B) C (if
Input: if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: S -> if (C) B D

Match: int ID (float ID) { while (ID <= NUM) { if
Stack: $ } Return Statement } Statement D B) C (
Input: (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 if

Match: int ID (float ID) { while (ID <= NUM) { if (
Stack: $ } Return Statement } Statement D B) C
Input: ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 (

Match: int ID (float ID) { while (ID <= NUM) { if (
Stack: $ } Return Statement } Statement D B) Keyword CompareOP Keyword
Input: ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: C -> Keyword CompareOP Keyword

Match: int ID (float ID) { while (ID <= NUM) { if (
Stack: $ } Return Statement } Statement D B) Keyword CompareOP ID
Input: ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: Keyword -> ID

Match: int ID (float ID) { while (ID <= NUM) { if (ID
Stack: $ } Return Statement } Statement D B) Keyword CompareOP
Input: == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 ID

Match: int ID (float ID) { while (ID <= NUM) { if (ID
Stack: $ } Return Statement } Statement D B) Keyword ==
Input: == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: CompareOP -> ==

Match: int ID (float ID) { while (ID <= NUM) { if (ID ==
Stack: $ } Return Statement } Statement D B) Keyword

Input: NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; }
$
Action: 匹配 ==

Match: int ID (float ID) { while (ID <= NUM) { if (ID ==
Stack: $ } Return Statement } Statement D B) NUM
Input: NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; }
$
Action: 输出: Keyword -> NUM

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM
Stack: $ } Return Statement } Statement D B)
Input:) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 NUM

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM)
Stack: $ } Return Statement } Statement D B
Input: { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配)

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM)
Stack: $ } Return Statement } Statement D } Statement {
Input: { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: B -> { Statement }

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) {
Stack: $ } Return Statement } Statement D } Statement
Input: ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 {

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) {
Stack: $ } Return Statement } Statement D } Statement S
Input: ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: Statement -> S Statement

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) {
Stack: $ } Return Statement } Statement D } Statement N Keyword = ID
Input: ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: S -> ID = Keyword N

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID
Stack: $ } Return Statement } Statement D } Statement N Keyword =
Input: = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 ID

```
Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID =
Stack:      $ } Return Statement } Statement D } Statement N Keyword
Input:      NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:     匹配 =

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID =
Stack:      $ } Return Statement } Statement D } Statement N NUM
Input:      NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:     输出: Keyword -> NUM

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM
Stack:      $ } Return Statement } Statement D } Statement N
Input:      ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:     匹配 NUM

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM
Stack:      $ } Return Statement } Statement D } Statement ;
Input:      ; } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:     输出: N -> ;

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ;
Stack:      $ } Return Statement } Statement D } Statement
Input:      } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:     匹配 ;

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ;
Stack:      $ } Return Statement } Statement D }
Input:      } else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:     输出: Statement -> ε

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; }
Stack:      $ } Return Statement } Statement D
Input:      else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:     匹配 }

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; }
Stack:      $ } Return Statement } Statement B else
Input:      else { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action:     输出: D -> else B

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
Stack:      $ } Return Statement } Statement B
Input:      { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
```

Action: 匹配 else

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
Stack: $ } Return Statement } Statement } Statement {
Input: { int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: B -> { Statement }

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else {
Stack: $ } Return Statement } Statement } Statement
Input: int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 {

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else {
Stack: $ } Return Statement } Statement } Statement S
Input: int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: Statement -> S Statement

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else {
Stack: $ } Return Statement } Statement } Statement M ID DataType
Input: int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: S -> DataType ID M

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else {
Stack: $ } Return Statement } Statement } Statement M ID int
Input: int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: DataType -> int

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int
Stack: $ } Return Statement } Statement } Statement M ID
Input: ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 int

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID
Stack: $ } Return Statement } Statement } Statement M
Input: = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 ID

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID
Stack: $ } Return Statement } Statement } Statement ; NUM =
Input: = NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 输出: M -> = NUM ;

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID =
Stack: $ } Return Statement } Statement } Statement ; NUM
Input: NUM ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 =

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM
Stack: $ } Return Statement } Statement } Statement ;
Input: ; ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 NUM

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ;
Stack: $ } Return Statement } Statement } Statement
Input: ID = ID + NUM ; } } return NUM ; } $
Action: 匹配 ;

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ;
Stack: $ } Return Statement } Statement } Statement S
Input: ID = ID + NUM ; } } return NUM ; } $
Action: 输出: Statement -> S Statement

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ;
Stack: $ } Return Statement } Statement } Statement N Keyword = ID
Input: ID = ID + NUM ; } } return NUM ; } $
Action: 输出: S -> ID = Keyword N

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID
Stack: $ } Return Statement } Statement } Statement N Keyword =
Input: = ID + NUM ; } } return NUM ; } $
Action: 匹配 ID

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID =
Stack: $ } Return Statement } Statement } Statement N Keyword
Input: ID + NUM ; } } return NUM ; } $
Action: 匹配 =

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else

{ int ID = NUM ; ID =
Stack: $ } Return Statement } Statement } Statement N ID
Input: ID + NUM ; } } return NUM ; } $
Action: 输出: Keyword -> ID

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID = NUM ; ID = ID
Stack: $ } Return Statement } Statement } Statement N
Input: + NUM ; } } return NUM ; } $
Action: 匹配 ID

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID = NUM ; ID = ID
Stack: $ } Return Statement } Statement } Statement ; Keyword OP
Input: + NUM ; } } return NUM ; } $
Action: 输出: N -> OP Keyword ;

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID = NUM ; ID = ID
Stack: $ } Return Statement } Statement } Statement ; Keyword +
Input: + NUM ; } } return NUM ; } $
Action: 输出: OP -> +

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID = NUM ; ID = ID +
Stack: $ } Return Statement } Statement } Statement ; Keyword
Input: NUM ; } } return NUM ; } $
Action: 匹配 +

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID = NUM ; ID = ID +
Stack: $ } Return Statement } Statement } Statement ; NUM
Input: NUM ; } } return NUM ; } $
Action: 输出: Keyword -> NUM

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM
Stack: $ } Return Statement } Statement } Statement ;
Input: ; } } return NUM ; } $
Action: 匹配 NUM

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ;
Stack: $ } Return Statement } Statement } Statement

Input: } } return NUM ; } $
Action: 匹配 ;

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ;
Stack: $ } Return Statement } Statement }
Input: } } return NUM ; } $
Action: 输出: Statement -> ε

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; }
Stack: $ } Return Statement } Statement
Input: } return NUM ; } $
Action: 匹配 }

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; }
Stack: $ } Return Statement }
Input: } return NUM ; } $
Action: 输出: Statement -> ε

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } }
Stack: $ } Return Statement
Input: return NUM ; } $
Action: 匹配 }

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } }
Stack: $ } Return
Input: return NUM ; } $
Action: 输出: Statement -> ε

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } }
Stack: $ } ; Keyword return
Input: return NUM ; } $
Action: 输出: Return -> return Keyword ;

Match: int ID (float ID) { while (ID <= NUM) { if (ID == NUM) { ID = NUM ; } else { int ID = NUM ; ID = ID + NUM ; } } return
Stack: $ } ; Keyword
Input: NUM ; } $
Action: 匹配 return

```
Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ; } } return
Stack:      $ } ; NUM
Input:      NUM ; } $
Action:     输出: Keyword -> NUM

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ; } } return NUM
Stack:      $ } ;
Input:      ; } $
Action:     匹配 NUM

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ; } } return NUM ;
Stack:      $ }
Input:      } $
Action:     匹配 ;

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ; } } return NUM ; }
Stack:      $
Input:      $
Action:     匹配 }

Match:      int ID ( float ID ) { while ( ID <= NUM ) { if ( ID == NUM ) { ID = NUM ; } else
{ int ID = NUM ; ID = ID + NUM ; } } return NUM ; } $
Stack:
Input:
Action:     匹配 $
```

9. 问题和解决方案

这次实验的问题主要是验证几个核心算法的正确性，在我的程序中，核心算法写的比较乱，自己也很容易绕晕，所以在验证上有一定的困难。但最后还是耐心下来和龙书给出的算法校对，最后应该没有什么问题。

10. 实验感受和评价

个人感觉这次实验比之前的词法分析要难，难在该用何种数据结构或者类去表示语法分析中的一些对象以及核心算法也具有一定的挑战性，特别是从自然语言描述转化到程序语言这个过程。个人认为这次完成的语法分析器只是一个简单版本，还不支持二义文法，文法的定义也比较少，只能算是一个小程序。