『编译技术』SysY-Mips编译器设计——语法分析

章节目录

零. 任务目标

设计并实现语法分析程序，分析语法成分，建立正确的语法树。

一. 终结符

- Ident
- Number
- FormatString（即使它有文法，仍将其看作终结符）
- 各种符号，关键字

二. 非终结符

将在下列出非终结符的FIRST以及是否有冲突情况，以便于解决后续问题；从简单至复杂分析，有助于分析复杂非终结符的FIRST。

非终结符	FIRST集	备注
`<CompUnit>`	`{}`	
`<LVal>`	`{Ident}`	
`<PrimaryExp>`	`{'(', Number, Ident}`	
`<UnaryOp>`	`{'+', '-', '!'}`	
`<UnaryExp>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<MulExp>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<AddExp>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<ConstExp>`	`{'(', Number, Ident, '+', '-', '!'}`	特殊: Ident只能是常量
`<Exp>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<ForStmt>`	`{Ident}`	
`<FuncRParams>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<RelExp>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<EqExp>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<LAndExp>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<LOrExp>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<Cond>`	`{'(', Number, Ident, '+', '-', '!'}`	
`<Block>`	`{'{'}`	
`<Stmt>`	`{'if', 'for', 'break', 'continue', 'return', Ident, 'printf', '{', ';', '(', Number, '+', '-', '!'}`	**存在冲突**
`<ConstDecl>`	`{'const'}`	
`<ConstDef>`	`{Ident}`	
`<ConstInitVal>`	`{'(', Number, Ident, '+', '-', '!', '{'}`	
`<BType>`	`{'int'}`	可能要增加基本类型，因此不能将其视为终结符
`<VarDecl>`	`{'int'}`	
`<Decl>`	`{'int', 'const'}`	
`<VarDef>`	`{Ident}`	**存在冲突**
`<Initial>`	`{'(', Number, Ident, '+', '-', '!', '{'}`	

非终结符	FIRST集	备注
`<FuncType>`	`{'void', 'int'}`	
`<FuncDef>`	`{'void', 'int'}`	
`<MainFuncDef>`	`{'int'}`	
`<FuncFParam>`	`{'int'}`	
`<FuncFParams>`	`{'int'}`	
`<BlockItem>`	`{'const', 'int', 'if', 'for', 'break', 'continue', 'return', Ident, 'printf', '{', ';', '(', Number, '+', '-', '!'}`	
`<CompUnit>`	`{'int', 'const', 'void'}`	**存在冲突**

三. 解决非终结符具有多产生式的问题

由上述分析可以得到总共有**四**处非终结符的FIRST存在冲突，接下来依次给出解决方案：

语句冲突

```
FrontEnd.NonTerminal.AllStmt → LVal '=' Exp ';'
    | [Exp] ';'
    | Block
    | 'if' '(' Cond ')' AllStmt [ 'else' AllStmt ]
    | 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' AllStmt
    | 'break' ';'
    | 'continue' ';'
    | 'return' [Exp] ';'
    | LVal '=' 'getint''('')'';'
    | 'printf''('FormatString{','Exp}')'';'
```

主要体现分支中两个`LVal`和`Exp`的 FIRST 集合存在交集{`Ident`}，若当前词法分析的单词为`Ident`，我们就无法判断该从哪一个分支进行解析。考虑到`Exp`可以推理出`LVal`（`Exp` ➡ `AddExp` ➡ `MulExp` ➡ `UnaryExp` ➡ `PrimaryExp` ➡ `LVal`），因此我们可以用`Exp`的解析方法来解析`LVal`。如果当前单词为`Ident`，则

- 首先利用调用`Exp`的子程序来解析出语法成分`Exp`，判断下一个单词是`';'`还是`'='`，如果是`';'`，则按第三条产生式处理，完成Stmt解析，否则转第2步，从前两条产生式中选择一条解析
- 从`Exp`提取出`LVal`（该`Exp`一定由唯一的`LVal`组成），继续判断下一个单词是不是`'getint'`，如果是则按第二条产生式处理，否则按第一条产生式处理，完成`Stmt`解析

变量定义冲突

```
VarDef -> Ident { '[' ConstExp ']' }
 | Ident { '[' ConstExp ']' } '=' InitVal
```

两分支 FIRST 存在交集{`Ident`}，不难发现前半部分都相同，因此改写文法为：

```
VarDef -> Ident { '[' ConstExp ']' } [ '=' InitVal ]
```

编译单元冲突

```
CompUnit -> { Decl } { FuncDef } MainFuncDef
```

虽然其中没有分支，但是也算是分支文法的改写形式，`Decl`，`FuncDef`，`MainFuncDef`的FIRST集存在交集{`'int'`}

解决方法，使用**未来探测法**：

- 若当前读取到的词是`int`，则继续预读下一个单词，若是`main`，则回退并进入主函数解析；若都为变量名，则跳转第二步
- 再读取下一个单词，若是`'('`则跳转到函数定义解析，若为其他(具体来说包含`'='`,`';'`,`'['`)则跳转到变量声明解析中，别忘了此时要**回退两个单词**!

一元表达式冲突

```
UnaryExp -> PrimaryExp | Ident '(' [FuncRParams] ')'
```

`PrimaryExp`和`Ident`的FIRST集合存在交集{`Ident`}，因此当首单词为`Ident`时仍使用未来探测法来预读下一个单词，若为`'('`则进入第二分支进行解析，反之回退，并进入第一分支进行解析。

四. 左递归文法的改写

通过理论课程的学习，我们知道左递归文法是无法被自顶向下分析解析的，因为会无限递归，因此对于左递归文法需要改写。

左递归文法全部出现在各类表达式文法当中，统计并改写如下：

- 乘除模表达式

```
MulExp -> UnaryExp { ('*' | '/' | '%') UnaryExp }
```

- 加减表达式

```
AddExp -> FrontEnd.NonTerminal.MulExp { ('+' | '-') MulExp }
```

- 关系表达式

```
RelExP -> AddExp { ('<' | '>' | '<=' | '>=') AddExp }
```

- 相等表达式

```
EqExp -> RelExp { ('==' | '!=') RelExp }
```

- 逻辑与表达式

```
LAndExp -> EqExp { '&&' EqExp }
```

- 逻辑或表达式

```
LOrExp -> LAndExp { '||' LAndExp }
```

注意：我们为了能正确分析而改写了文法，但是输出结果应与原文法保持相同，例如`2+3`，使用改写后文法将`+`两边都看作`MulExp`，然而按照实际文法中，`+`左边为`AddExp`，`+`右边为`MulExp`。

解决方法：可以在每次解析(`'+'` | `'-'`) `MulExp`之前，先将之前已经解析出的若干个`MulExp`合成一个`AddExp`，输出一次`<AddExp>`。

五. 规定

为了确保语法分析和词法分析的配合，我们做出如下规定供参考：

- 一个子程序在调用其他子程序前，需要调用词法分析器来预读一个单词
- 一个子程序在退出时，需要调用词法分析器来预读一个单词

有了上述规定，就可以确保：

- 刚进入一个子程序时，词法分析器已经预读好了一个单词
- 从一个子程序返回时，词法分析器已经预读好了一个单词

再规定：

- 进入一个解析函数中，先创建该类实例，再按流程逐步填充对象，最后整体返回

六. 语法分析Bug

- 先新建Parser类导致Lexer提前预读，致使注释无法正常清除，应当先清注释再`new Parser()`；
- 定义时`Ident`后面Follow的终结符情况未考虑完全，少考虑了一种如`int i, j;`带有逗号的情况。