PA1-B实验报告

计73 王焱 2017050024

一、实验综述

基于 LL(1) 的语法分析与错误恢复 。任务与 PA1-A 相同，在已有的框架上加入三个新的语法特性，实现 Decaf 语言编译器的词法分析和语法分析部分，同时生成抽象语法树。但采用自顶向下的语法分析方法，并要求支持一定程度的错误恢复。

二、具体实现

本次实验在PA1-A的基础上加以修改，词法分析部分沿用PA1-A 的实现。

错误恢复

采用实验指导书上介绍的算法：

与应急恢复的方法类似，当分析非终结符A时，若当前输入符号a∉Begin(A)，则先报错， 然后跳过输入符号串中的一些符号，直至遇到Begin(A)∪End(A)中的符号，若遇到的是Begin(A)中的符号，可恢复分析A；若遇到的是End(A)中的符号，则A分析失败，继续分析A后面的符号。

```
private SemValue parseSymbol(int symbol, Set<Integer> follow) {
            var result = query(symbol, token); // get production by lookahead
symbol
            // obtain the Begin and End set of current non-terminate symbol
            var begin = beginSet(symbol);
            var end = followSet(symbol);
            end.addAll(follow);

            if(!begin.contains(token)) {
                hasError = true;
                yyerror("syntax error");
                while (true){
                    if(begin.contains(token)){
                        result = query(symbol, token);
                        break;
                    }else if(end.contains(token)){
                        return null;
                    }
                    token = nextToken();
                }
            }
            var actionId = result.getKey(); // get user-defined action
            var right = result.getValue(); // right-hand side of production
            var length = right.size();
            var params = new SemValue[length + 1];

            for (var i = 0; i < length; i++) {// parse right-hand side symbols
one by one
                var term = right.get(i);
                params[i + 1] = isNonTerminal(term)
```

```
                        ? parseSymbol(term, end)//for non terminals: recursively
parse it
                        : matchToken(term) // for terminals: match token
                ;
            }
            params[0] = new SemValue();
            if (!hasError)
                act(actionId, params); // do user-defined action
            return params[0];
        }
```

改写为LL1文法

1.abstract

由于新的产生式右端的第一个是新添加的token，直接添加相应的产生式就是LL1文法，所以只需要按照tree.java 中构造函数的实现，分别修改 new ClassDef 和 new MethodDef

```
ClassDef        :   ABSTRACT CLASS Id ExtendsClause '{' FieldList '}'
                    {
                        $$ = svClass(new ClassDef($3.id,
Optional.ofNullable($4.id), $6.fieldList, $2.pos, true));
                    }
                |   CLASS Id ExtendsClause '{' FieldList '}'
                    {
                        $$ = svClass(new ClassDef($2.id,
Optional.ofNullable($3.id), $5.fieldList, $1.pos, false));
                    }
                ;
```

```
FieldList       :   ABSTRACT Type Id '(' VarList ')' ';' FieldList
                    {
                        $$ = $8;
                        $$.fieldList.add(0, new MethodDef(true, false, $3.id,
$2.type, $5.varList, null, $3.pos));
                    }
                |   STATIC Type Id '(' VarList ')' Block FieldList
                    {
                        $$ = $8;
                        $$.fieldList.add(0, new MethodDef(false, true, $3.id,
$2.type, $5.varList, $7.block, $3.pos));
                    }
                |   Type Id AfterIdField FieldList
                    {
                        $$ = $4;
                        if ($3.varList != null) {
                            $$.fieldList.add(0, new MethodDef(false, false,
$2.id, $1.type, $3.varList, $3.block, $2.pos));
                        } else {
                            $$.fieldList.add(0, new VarDef($1.type, $2.id,
$2.pos));
                        }
                    }
                |   /* empty */
                    {
```

```
                        $$ = svFields();
                    }
                ;
```

2.var

由于新的产生式右端的第一个是新添加的token，直接添加相应的产生式就是LL1文法。

```
SimpleStmt      :   Var Initializer
                    {
                        $$ = svStmt(new LocalVarDef($1.type, $1.id, $2.pos,
Optional.ofNullable($2.expr), $1.pos));
                    }
                |   Expr Initializer
                    {
                        if ($2.expr != null) {
                            if ($1.expr instanceof LValue) {
                                var lv = (LValue) $1.expr;
                                $$ = svStmt(new Assign(lv, $2.expr, $2.pos));
                            } else {
                                yyerror("syntax error");
                            }
                        } else {
                            $$ = svStmt(new ExprEval($1.expr, $1.pos));
                        }
                    }
                |   VAR Id '=' Expr
                    {
                        $$ = svStmt(new LocalVarDef(null, $2.id, $3.pos,
Optional.ofNullable($4.expr), $2.pos));
                    }
                |   /* empty */
                    {
                        $$ = svStmt(null);
                    }
                ;
```

3.First-class Functions

（1）函数类型

仿照原来的ArrayType 写FunType，并写为LL1文法。TypeList改为左结合。

```
Type            :   AtomType ArrayFunType
                    {
                        $$ = $1;
                        for (var sv: $2.thunkList) {
                            if(sv.typeList != null) {
                                $$.type = new TLambda($1.type, sv.typeList,
$1.pos);
                            }
                            else {
                                $$.type = new TArray($$.type, $1.type.pos);
                            }
                        }
                    }
                ;
```

```
ArrayFunType        :   '[' ']' ArrayFunType
                        {
                            var sv = new SemValue();
                            $$ = $3;
                            $$.thunkList.add(0, sv);
                        }
                    |   '(' TypeList ')' ArrayFunType
                        {
                            var sv = new SemValue();
                            sv.typeList = $2.typeList;
                            $$ = $4;
                            $$.thunkList.add(0, sv);
                        }
                    |   /* empty */
                        {
                            $$ = new SemValue();
                            $$.thunkList = new ArrayList<>();
                        }
                    ;


TypeList            :   Type TypeList1
                        {
                            $$ = $2;
                            $$.typeList.add(0, $1.type);
                        }
                    |   /* empty */
                        {
                            $$ = svTypes();
                        }
                    ;


TypeList1           :   ',' Type TypeList1
                        {
                            $$ = $3;
                            $$.typeList.add(0, $2.type);
                        }
                    |   /* empty */
                        {
                            $$ = svTypes();
                        }
                    ;
```

(2) lambda表达式

提取左公因子

```
Expr                :   Expr1
                        {
                            $$ = $1;
                        }
                    |   FUN '(' VarList ')' Lambda
                        {
```

```
                        $$ = svExpr(new Lambda($3.varList, $5.expr, $5.block,
$1.pos));
                    }
                ;
Lambda          :   Block
                    {
                        $$ = $1;
                    }
                |   EQUAL_GREATER Expr
                    {
                        $$ = $2;
                    }
                ;
Expr1           :   Expr2 ExprT1
                    {
                        $$ = buildBinaryExpr($1, $2.thunkList);
                    }
                ;
```

提取左公因子后，需要有一个一致的实例化，所以修改在`tree.java`中`LambdaBlock`和`LambdaExpr`类，将其合写为一个`Lambda`类。

```
    public static class Lambda extends Expr
    {
        // Tree elements
        public List<LocalVarDef> params;
        public Expr expr;
        public Block body;
        // For convenience
        public String name;

        public Lambda(List<LocalVarDef> params, Expr expr, Block body, Pos pos)
{
            super(Kind.LAMBDA, "Lambda", pos);
            this.params = params;
            this.expr = expr;
            this.body = body;
        }

        @Override
        public Object treeElementAt(int index) {
            return switch (index) {
                case 0 -> params;
                case 1 -> ( expr == null ) ? body : expr;
                default -> throw new IndexOutOfBoundsException(index);
            };
        }

        @Override
        public int treeArity() {
            return 2;
        }

        @Override
```

```
        public <C> void accept(Visitor<C> v, C ctx) {
            v.visitLambda(this, ctx);
        }

    }
```

（3）函数调用

首先根据Call的构造函数的实现，修改decaf.spec中的new Call语句，然后改成LL1文法。

原来的文法为：ExprT8 ->'[' Expr ']' ExprT8 | '.' Id ExprListOpt ExprT8 |epsilon

需要添加产生式： ExprT8 -> '(' ExprList ')' ExprT8

由于ExprListOpt 可推出 '(' ExprList ')'，会产生重复，所以删除ExprListOpt，并且Expr9中只留下 Id。

```
ExprT8          :   '[' Expr ']' ExprT8
                    {
                        var sv = new SemValue();
                        sv.expr = $2.expr;
                        sv.pos = $1.pos;

                        $$ = $4;
                        $$.thunkList.add(0, sv);
                    }
                |   '.' Id ExprT8
                    {
                        var sv = new SemValue();
                        sv.id = $2.id;
                        sv.pos = $2.pos;

                        $$ = $3;
                        $$.thunkList.add(0, sv);
                    }
                |   '(' ExprList ')' ExprT8
                    {
                        var sv =new SemValue();
                        sv.exprList = $2.exprList;
                        sv.pos =$1.pos;
                        $$ =$4;
                        $$.thunkList.add(0,sv);
                    }
                |   /* empty */
                    {
                        $$ = new SemValue();
                        $$.thunkList = new ArrayList<>();
                    }
                ;

Expr9           :
                //////
                |   Id
                    {
                        $$ = svExpr(new VarSel($1.id, $1.pos));
                    }
                ;
```

三、实验思考

Q1. 本阶段框架是如何解决空悬 else (dangling-else) 问题的?

答：强制规定else的优先级高，从而优先移入else，使else匹配最近的if。

```
ElseClause      :   ELSE Stmt // higher priority, which triggers a warning (that
is expected)
                    {
                        $$ = $2;
                    }
                |   /* empty */
                    {
                        $$ = svStmt(null);
                    }
                ;
```

Q2. 使用 LL(1) 文法如何描述二元运算符的优先级与结合性？请结合框架中的文法，举例说明。

答：优先级是通过定义非终结符的推导关系确定的，如框架代码中，加法和乘法的实现，Op5为加法，Op6为乘法。

```
Expr5           :   Expr6 ExprT5
ExprT5          :   Op5 Expr6 ExprT5|epsilon
Expr6           :   Expr7 ExprT6
ExprT6          :   Op6 Expr7 ExprT6|epsilon
```

同级运算符的结合性是通过产生式中非终结符的位置确定的，如框架代码中，加法的结合性为左结合。

```
ExprT5          :   Op5 Expr6 ExprT5
```

Q3. 无论何种错误恢复方法，都无法完全避免误报的问题。 请举出一个具体的Decaf 程序（显然它要有语法错误），用你实现的错误恢复算法进行语法分析时会带来误报。 并说明该算法为什么无法避免这种误报。

答：

```
calss Main{
    abstract void foo(){}
    static void mian(){}
}
```

第二行 { 匹配失败，main的 } 与Main的 { 匹配。

当产生匹配失败时，尝试继续匹配，无法保证这条语句的结尾之前可以继续匹配，有可能会影响下一句，产生误报。

四、实验小结

本次实验开始，为了以后方便，我将框架换到了完整框架，手动merge还是出了错，后来用助教推荐的meld软件又查了一遍才可以，感谢助教。这次试验错误恢复，参考实验指导和往年代码实现，相对容易一些，改写LL1文法实现起来还是比较困难，思路是参考框架中已有的实现，比如加法和乘法，比如数组类型，当然还有大佬们的指点，感谢。