编译原理PA1B实验报告

张钰晖 计55 2015011372 yuhui-zh15@mails.tsinghua.edu.cn

任务描述

PA1A中，我们借助LEX和YACC完成了Decaf的词法、语法分析。

本阶段任务与PA1A相同，但不再使用YACC，而是手工实现自顶而下的语法分析，并支持一定程度的错误恢复。

本阶段需要借助词法分析工具JFlex和语法分析工具BYACC。

文件说明

在本阶段，以下文件非常重要，主要需要修改以下文件。

文件名	含义	说明
BaseLexer.java	词法分析程序基础	沿用PA1A的改动
Lexer.java	词法分析器	沿用PA1A自动生成的程序
Parser.spec	LL(1)文法描述	增加新特性对应的LL(1)文法
Parser.java	语法分析器	增加错误恢复功能
Tree.java	抽象语法树各种节点	新增抽象语法树节点所需的类

- 词法分析阶段：沿用PA1A的结果
  - BaseLexer.java根据PA1A进行修改
  - Lexer.java沿用PA1A自动生成的程序
- 语法分析阶段：修改Parser.spec、Parser.java和Tree.java实现
  - Parsel.spec为LL(1)文法，其每个非终结符均为SemValue类，SemValue类是Tree各种类的整合体（类似指针）
  - Parser.java实现了语法分析功能，需要增加错误恢复功能
  - Tree.java文件主要包含了抽象语法树各种节点

单词符号说明

- 关键字：语言保留字，如"int", "void"...
- 标识符：字母开头，后跟字母数字下划线，如"my_var123"...
- 常量：整数、布尔、字符串，如"123"...
- 算符和界符：单字符和双字符，如"+", "-"...
- 注释：单行注释，"//"开头

实验内容及实现

本次实验分为三个阶段，主要修改列于表格之中。

步骤一：阅读LL(1)分析算法的实现

本阶段无需修改文件。

框架提供了通用的parse()函数实现了递归下降的LL(1)分析方法：

SemValue parse(int symbol, Set<Integer> follow)

其中symbol为待分析的非终结符。若分析成功，则返回值存储了symbol所对应AST结点的值；若分析失败，则返回null。

其大致思路是，遇到非终结符则递归调用parse()函数；遇到终结符则调用matchToken()函数。

该函数实现思路清晰简洁，在此阶段无需修改，之后需增加错误恢复功能。

```
private SemValue parse(int symbol, Set<Integer> follow) {
    Pair<Integer, List<Integer>> result = query(symbol, lookahead); // get production by lookahead symbol
    int actionId = result.getKey(); // get user-defined action
    List<Integer> right = result.getValue(); // right-hand side of production
    int length = right.size();
    SemValue[] params = new SemValue[length + 1];
    for (int i = 0; i < length; i++) { // parse right-hand side symbols one by one
        int term = right.get(i);
        params[i + 1] = isNonTerminal(term)
                ? parse(term, follow) // for non terminals: recursively parse it
                : matchToken(term) // for terminals: match token
                ;
    }
    params[0] = new SemValue(); // initialize return value
    act(actionId, params); // do user-defined action
    return params[0];
}
```

步骤二：增加错误恢复功能

本阶段修改文件如下：

文件名	修改
Parser.java	修改函数parse()，增加错误恢复功能

在读懂parse()函数后，需要修改该函数使其支持错误恢复功能。

错误恢复功能是指，当输入的Decaf程序出现语法错误时，它还能对后续的程序继续分析，直至读到文件尾。课件中介绍了应急恢复和短语层恢复的方法，这里采用一种介于二者之间的错误恢复方法：

当分析非终结符A时，若当前输入符号$a \notin Begin(A)$，则先报错，然后跳过输入符号串中的一些符号，直至遇到$Begin(A) \cup End(A)$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，返回null，继续分析A后面的符号。

其中，为了少跳过一些符号与避免死循环，

- $Begin(A) = \{s|M[A,s] \neq \phi\}$（其中，$M$为预测分析表）
- $End(A) = Follow(A) \cup F$（其中，$F$为parse函数传入的第二个参数）

当匹配终结符失败时，只报错，但不消耗此匹配失败的终结符，而是将它保留在剩余输入串中。

修改的增加错误恢复功能的parse函数如下。

```
private SemValue parse(int symbol, Set<Integer> follow) {
    Set<Integer> begin = beginSet(symbol); // begin set as defined
    Set<Integer> end = new HashSet<Integer>(); // end set as defined
    end.addAll(followSet(symbol));
    end.addAll(follow);
    Pair<Integer, List<Integer>> result = query(symbol, lookahead); // get production by lookahead symbol
    if (result == null) {
        error();
        while (true) {
            if (begin.contains(lookahead)) return parse(symbol, follow);
            if (end.contains(lookahead)) return null;
            lookahead = lex();
        }
    } // error handler
    int actionId = result.getKey(); // get user-defined action
    List<Integer> right = result.getValue(); // right-hand side of production
    int length = right.size();
    SemValue[] params = new SemValue[length + 1];
    for (int i = 0; i < length; i++) { // parse right-hand side symbols one by one
        int term = right.get(i);
        params[i + 1] = isNonTerminal(term)
                ? parse(term, end) // for non terminals: recursively parse it
                : matchToken(term) // for terminals: match token
                ;
    }
    params[0] = new SemValue(); // initialize return value
    try { act(actionId, params); } catch (NullPointerException e) {} // do user-defined action, catch exception
    return params[0];
}
```

步骤三：增加新特性对应的LL(1)文法

本阶段修改文件如下：

文件名	修改
BaseLexer.java	新增函数imgConst()识别复数虚部
Lexer.java	沿用PA1A程序
Parser.spec	新增终结符"COMPLEX","@", "$", "#","PRINTCOMP","CASE","DEFAULT",":","SUPER","DCOPY","SCOPY","DO","OD","|||"
	新增规则SimpleType ::= COMPLEX
	新增规则Oper8 ::= @ | $ | #
	修改规则Expr7 ::= Oper7 ExprT7 | ExprT7
	新增规则ExprT7 ::= Oper8 Expr8 | Expr8
	新增规则Stmt ::= PrintCompStmt;
	新增规则PrintCompStmt ::= PRINTCOMP(ExprList)
	新增规则Expr9 ::= CASE (Expr) {ACaseExprList DefaultExpr}
	新增规则ACaseExprList ::= ACaseExpr ACaseExprList | 空
	新增规则ACaseExpr ::= Constant: Expr;
	新增规则DefaultExpr ::= DEFAULT: Expr;
	新增规则Expr9 ::= SUPER
	新增规则Expr9 ::= DCOPY(Expr) | SCOPY(Expr)
	新增规则Stmt ::= DoStmt;
	新增规则DoStmt ::= DO DoSubStmt DoBranchList OD
	新增规则DoBranchList ::= DoBranch DoBranchList | 空
	新增规则DoBranch ::= ||| DoSubStmt
	新增规则DoSubStmt ::= Expr : Stmt
Tree.java	修改类Unary，对运算符"@", "$", "#"支持
	修改类Literal，对复数虚部常量支持
	新增类PrintComp，对复数打印语句支持
	新增类CaseExpr，对case表达式支持
	新增类ACaseExpr，对case表达式单条语句支持

	新增类DefaultExpr，对case表达式default语句支持
	新增类SuperExpr，对super表达式支持
	新增类DcopyExpr和ScopyExpr，支持对象复制
	新增类DoStmt，对串行循环卫士支持
	新增类DoSubStmt，对串行循环卫士子语句支持

正确实现LL(1)分析算法后，在本阶段中我们将添加新的文法，调用工具自动完成Table类数据的更新，从而实现对新特性的支持。

在本阶段，支持的新的文法特性如下：

- 整复数类型的支持：本阶段需要增加整复数类型，增加识别复数常量虚部功能，增加获取复数实部、虚部及强制转换复数表达式，增加复数打印语句。
- case表达式的支持：本阶段需要支持case表达式，语法为case(表达式) {常量1:表达式1,..., default:表达式}。
- super表达式的支持：本阶段需要支持super表达式，类似this表达式。
- 对象复制的支持：本阶段需要支持深复制dcopy()和浅复制scopy()表达式。
- 串行循环卫士的支持：本阶段需要支持串行循环卫士语句，语法为do E1:S1|||E2:S2... od

技巧心得

1. 特殊产生式的支持

LL(1)文法不含左递归，可通过以下方式实现。（与YACC不同，YACC推荐左递归）

产生式	实现
A*	B ::= A B | 空
A+	B ::= A B | A
A?	B ::= A | 空

2. 消除冲突

在PA1A中实现的串行循环卫士语句不是LL(1)文法，需要对其文法进行变换，才可消除冲突。

- 原始文法如下：
  - Stmt ::= DoStmt ;
  - DoStmt ::= do DoBranch* DoSubStmt od
  - DoBranch ::= DoSubStmt |||
  - DoSubStmt ::= Expr : Stmt
- 变换后文法如下：
  - Stmt ::= DoStmt ;

    - DoStmt ::= do DoSubStmt DoBranch* od
    - DoBranch ::= ||| DoSubStmt
    - DoSubStmt ::= Expr : Stmt

通过这样的变换，全部文法中仅剩Else语法会产生冲突。

- ElseClause ::= ELSE Stmt
- ElseClause ::= <empty>

问题阐述

1. Else冲突

Decaf语言由于允许if语句的else分支为空，因此不是严格的LL(1)语言，但是我们的工具依然可以处理这种冲突。请根据工具所生成的预测分析表中if语句相关项的预测集合先做猜测，并对照工具wiki（https://github.com/paulzfm/LL1-Parser-Gen/wiki/2.-Strict-Mode），理解本工具的处理方法。请在实验报告中说明此方法的原理，并举一个你自己构造的例子加以说明。

(1) Decaf语言产生式如下：

- IfStmt ::= IF '(' Expr ')' Stmt ElseClause
- ElseClause ::= ELSE Stmt | <empty>

由于PS(ElseClause→ELSE Stmt)={ELSE}, PS(ElseClause→<empty>)={ELSE, PRINT, CASE, COMPLEX, VOID, ...}

故PS(ElseClause→ELSE Stmt)={ELSE}∩PS(ElseClause→<empty>)={ELSE}$\neq \phi$

因此Decaf语言不是严格LL(1)语言。

(2) 工具所生成的预测分析表如下：

```
case ElseClause: {
    switch (lookahead) {
        case ELSE:
            return new Pair<>(50, Arrays.asList(ELSE, Stmt));
        case PRINT:
        case CASE:
        case COMPLEX:
        case VOID:
        case FOR:
        case '!':
        case '-':
        case CLASS:
        case PRINTCOMP:
        case READ_LINE:
        case WHILE:
        case RETURN:
        case NULL:
        case NEXT:
        case INT:
        case SCOPY:
        case '}':
        case '@':
        case DO:
        case IDENTIFIER:
        case NEW:
        case '$':
        case IF:
        case THIS:
        case INSTANCEOF:
        case STRING:
        case LITERAL:
        case '(':
        case ';':
        case '#':
        case OD:
        case DCOPY:
        case BOOL:
        case SUPER:
        case BREAK:
        case READ_INTEGER:
        case '{':
            return new Pair<>(131, Arrays.asList());
        default: return null;
    }
}
```

观察这段代码可以看出，工具认为ElseClause→ELSE Stmt优先级高于ElseClause→<empty>，因此仅将ELSE放入了PS(ElseClause→ELSE Stmt)集合，将ELSE移出了PS(ElseClause→<empty>)集合，此时PS(ElseClause→ELSE Stmt)={ELSE}∩PS(ElseClause→<empty>)=ϕ，从而解决了冲突，成为了LL(1)语法。

因此实际对应的Decaf语言产生式如下：

- IfStmt ::= IF '(' Expr ')' Stmt ElseClause
- ElseClause ::= ELSE Stmt **(high priority)** | <empty>

(3) 工具Wiki如下：

> An LL(1) grammar requires that for any two productions `A -> alpha` and `A -> beta` with the same left-hand side, the intersection of their predictive sets `PS(A -> alpha)` and `PS(A -> beta)` should be empty. In other words, the predictive lookahead symbols should never overlap. If they do, for instance, `C = PS(A -> alpha) & PS(A -> beta) = {'('}`, it is not possible to tell which production is taken when the lookahead token is '(', because both are available.
>
> Practically, some non-LL(1) grammar can be parsed in LL(1) fashion by explicitly assuming a precedence. For the instance above, we assign higher priority to the production `A -> alpha`, and when '(' is the lookahead symbol, `A -> alpha` rather than `A -> beta` will be applied by the parser. By modifying the predictive set for `A -> beta` as `PS(A -> beta)' = PS(A -> beta) - C`, we see that `PS(A -> alpha) & PS'(A -> beta)` is now empty and hence satisfies the definition of LL(1) grammar.
>
> Consider grammar `G[S]`:
>
> ```
> S -> if C then S E
> E -> else S | <empty>
> ```
>
> Grammar `G` is not LL(1) because `PS(E -> else S) & PS(E -> <empty>) = {else}`. Nonetheless, we can assign higher priority to `E -> else S` and parse a `G` program by LL(1) technique.

根据Wiki可知，笔者的猜想是正确的。Wiki指出，若两个产生式A -> alpha和A -> beta的预测集合交集不为空，假设A -> alpha的优先级更高，则将A -> alpha和A -> beta的预测集合的交集部分移出A -> beta的预测集合，从而使得预测集合的交集为空，满足LL(1)文法。

(4) 举例如下：

考虑以下输入：

```
if (true) if (false) Print("A"); else Print("B");
```

考虑Decaf产生式：

- IfStmt ::= IF '(' Expr ')' Stmt ElseClause
- ElseClause ::= ELSE Stmt | <empty>

若ElseClause ::= ELSE Stmt优先级更高，则语法解析为：

```
if (true) { if (false) Print("T"); else Print("F"); }
```

此时将输出F。

若ElseClause ::= <empty>优先级更高，则语法解析为：

```
if (true) { if (false) Print("T"); } else Print("F");
```

此时将无输出。

2. 错误误报

无论何种错误处理方法，都无法完全避免误报的问题。请举出一个语法错误的Decaf 程序例子，用你实现的Parser进行语法分析会带来误报。并说明为什么你用的错误处理方法无法避免这种误报。

(1) 举例如下：

考虑以下输入：

```
class Main {
    static int main() {
        return case(a) {
            default: 1;
            0: 0;
        };
    }
}
```

正确报错应为(以下结果为PA1A采用YACC分析的程序报错的结果)：

```
*** Error at (5,13): syntax error
```

实际报错为(以下结果为PA1B程序报错的结果)：

```
*** Error at (5,13): syntax error
*** Error at (5,14): syntax error
*** Error at (5,16): syntax error
*** Error at (6,10): syntax error
*** Error at (8,1): syntax error
```

(2) 分析如下：

在上例中，case表达式的产生式如下：

- Expr9 ::= CASE (Expr) {ACaseExprList DefaultExpr}
- ACaseExprList ::= ACaseExpr ACaseExprList | 空
- ACaseExpr ::= Constant: Expr;
- DefaultExpr ::= DEFAULT: Expr;

程序在分析case时分析流程如下，加粗为当前正在分析的非终结符：

步骤	产生式(粗体为当前分析终结符)	分析	位置
1	**CASE** (Expr) {ACaseExprList DefaultExpr}	Matchtoken(CASE)正确	(3, 16)
2	CASE **(**Expr) {ACaseExprList DefaultExpr}	Matchtoken('(')正确	(3, 20)
3	CASE (**Expr**) {ACaseExprList DefaultExpr}	parse(Expr)正确	(3, 21)
4	CASE (Expr**)** {ACaseExprList DefaultExpr}	Matchtoken(')')正确	(3, 22)
5	CASE (Expr) **{**ACaseExprList DefaultExpr}	Matchtoken('{')正确	(3, 24)
6	CASE (Expr) {**ACaseExprList** DefaultExpr}	parse(ACaseExprList)正确(采用产生式 ACaseExprList ::= 空)	(4, 13)
7	CASE (Expr) {ACaseExprList **DefaultExpr**}	parse(DefaultExprList)正确(采用产生式 DefaultExpr ::= DEFAULT: Expr;)	(4, 13)
8	CASE (Expr) {ACaseExprList DefaultExpr**}**	Matchtoken('}')**错误**，报错并分析失败，**递归返回**	**(5, 13)**
9	上一级递归分析	不断报错	...

报错的原因可以通过表格的分析流程清晰地看出，由于在分析失败case表达式后递归返回上一层，导致分析成功之后部分：

```
            0: 0;
        };
    }
}
```

不断被其父节点分析，而不是被当前节点分析，这会导致不断误报，而正确的分析报错应该仅仅报错一个位置。

这种错误处理方式之所以无法避免这种误报，是因为在分析失败后被迫返回上一层之后，会引起一系列连锁反应，最终导致不断误报。假设当前正在分析B，其递归返回节点在分析A，如果分析B失败被迫返回时，后面内容本应该被B分析，而却不得不断的被A分析时，便会造成一系列不当的报错，递归返回造成了一个不可避免的导火索。

总结

通过本次实验，笔者对语法分析的认识有了质的提高，不仅深入理解了语法分析，更清晰地理解了错误恢复的原理与其局限性。有了PA1A的基础，PA1B的实现不再那么痛苦，写完之后有一种豁然开朗之感。尤其是parse()函数，寥寥几行却完成了整个语法分析的核心使命，实在令人惊叹，笔者在实践之中真正感受到了编译的神奇之处。