**编译原理decaf/Mind PA1-B实验报告**

魏宇轩 2015011942

**1. 本阶段主要工作**

**1.1 修改parse函数，增加错误恢复功能**

递归下降函数算法流程（其中新增的算法逻辑用**加粗棕色字体**表示）

输入：当前非终结符A和辅助集合follow（用于计算followSet）

输出：非终结符A的解析结果

|  |
| --- |
| 根据当前非终结符A和下一个终结符a计算预测集PS  **保存当前的下一个终结符为a0**  如果PS为空：  报错  **按README的方法计算A的beginSet和followSet**  **循环读取下一个终结符直到读到一个终结符b属于beginSet或followSet**  **如果b属于beginSet:**  **根据A和b计算新的预测集PS’, 用PS’替代PS继续后续的分析**  **如果b属于followSet:**  **将下一个终结符重置为a0**  **返回null（放弃解析非终结符A）**  如果PS非空：  **用A的followSet更新辅助集合follow**  对PS中的每一个非终结符递归下降分析  返回分析结果 |

**1.2 修改Parser.spec，增加新语法特性，并使之符合LL(1)文法；根据Parser.spec的内容修改抽象语法树tree.Tree.java**

对象复制语句、sealed修饰的类定义、自动类型推导、Python风格数组comprehension表达式、数组迭代语句的文法自然符合LL(1)文法，直接参照README的参考实现即可，故在此不再详细展开。而串行条件卫士、数组常量、数组初始化常量、数组拼接、取子数组、数组下标动态访问这些语法特性README给出的参考含有左递归或左公因子，不符合LL(1)文法，需要进行修改。下文一一进行阐述。

**1.2.1 串行条件卫士**

（1）左公因子if：串行条件卫士语句和if语句有左公因子if, 引进

StmtWithIf := ‘(‘ Expr ‘)’ Stmt ElseClause /\* if语句 \*/

| ‘{‘ IfBlock ‘}’ /\* 串行条件卫士 \*/

消除左公因子if.

（2）左递归：多个条件语句（Expr ‘:’ Stmt）形成左递归，用如下语法消除左递归

IfBlock : IfSubStmt IfBranch

{

$$.guards = new ArrayList<Guard>();

$$.guards.add($1.guard);

if ($2.guards != null)

{

$$.guards.addAll($2.guards);

}

}

| /\* empty \*/

{

$$.guards = new ArrayList<Guard>();

}

;

IfBranch : PARGUARD IfSubStmt IfBranch

{

$$.guards = new ArrayList<Guard>();

$$.guards.add($2.guard);

if ($3.guards != null)

{

$$.guards.addAll($3.guards);

}

}

| /\* empty \*/

{

$$.guards = new ArrayList<Guard>();

}

;

IfSubStmt : Expr ':' Stmt

{

$$.guard = new Tree.Guard($1.expr, $3.stmt, $2.loc);

}

;

**1.2.2 数组常量**

消除左递归：

ArrayConstant : '[' ArrayConst ']'

{

$$.expr = new ArrayConst($2.elist, $2.loc);

}

;

ArrayConst : Constant Constants

{

$$.elist = new ArrayList<Tree.Expr>();

$$.elist.add($1.expr);

if ($2.elist != null)

{

$$.elist.addAll($2.elist);

}

}

| /\* empty \*/

{

$$.elist = new ArrayList<Tree.Expr>();

}

;

Constants : ',' Constant Constants

{

$$.elist = new ArrayList<Tree.Expr>();

$$.elist.add($2.expr);

if ($3.elist != null)

{

$$.elist.addAll($3.elist);

}

}

| /\* empty \*/

{

$$.elist = new ArrayList<Tree.Expr>();

}

;

**1.2.3 数组初始化常量和数组拼接**

数组初始化常量**%%**：左结合性引起左递归，要消除左递归

数组拼接**++**：右结合性引起左公因子，要消除左公因子

优先级：**+/-** > **%%** > **++** >**关系运算符**, 因此处理含有%%表达式和++表达式的代码应在处理关系运算符和+/-的代码段之间的位置

Expr4：含有关系运算符的表达式

ExprArrayAppend：含有++的表达式

ExprArrayCopy：含有%%的表达式

Expr5：含有+/-的表达式

Expr4 : ExprArrayAppend ExprT4

{

$$.expr = $1.expr;

if ($2.svec != null) {

for (int i = 0; i < $2.svec.size(); ++i) {

$$.expr = new Tree.Binary($2.svec.get(i), $$.expr,

$2.evec.get(i), $2.lvec.get(i));

}

}

}

;

ExprT4 : Oper4 ExprArrayAppend ExprT4

{

$$.svec = new Vector<Integer>();

$$.lvec = new Vector<Location>();

$$.evec = new Vector<Expr>();

$$.svec.add($1.counter);

$$.lvec.add($1.loc);

$$.evec.add($2.expr);

if ($3.svec != null) {

$$.svec.addAll($3.svec);

$$.lvec.addAll($3.lvec);

$$.evec.addAll($3.evec);

}

}

| /\* empty \*/

;

ExprArrayAppend : ExprArrayCopy ExprTaa

{

$$.expr = $1.expr;

if ($2.svec != null) {

for (int i = 0; i < $2.svec.size(); ++i) {

$$.expr = new Tree.Binary($2.svec.get(i), $$.expr,

$2.evec.get(i), $2.lvec.get(i));

}

}

}

;

ExprTaa : ARRAYAPPEND ExprArrayAppend

{

$$.svec = new Vector<Integer>();

$$.lvec = new Vector<Location>();

$$.evec = new Vector<Expr>();

$$.svec.add(Tree.ARRAYAPPEND);

$$.lvec.add($1.loc);

$$.evec.add($2.expr);

}

| /\* empty \*/

;

ExprArrayCopy : Expr5 ExprTac

{

$$.expr = $1.expr;

if ($2.svec != null) {

for (int i = 0; i < $2.svec.size(); ++i) {

$$.expr = new Tree.Binary($2.svec.get(i), $$.expr,

$2.evec.get(i), $2.lvec.get(i));

}

}

}

;

ExprTac : ARRAYCOPY Expr5 ExprTac

{

$$.svec = new Vector<Integer>();

$$.lvec = new Vector<Location>();

$$.evec = new Vector<Expr>();

$$.svec.add(Tree.ARRAYCOPY);

$$.lvec.add($1.loc);

$$.evec.add($2.expr);

if ($3.svec != null) {

$$.svec.addAll($3.svec);

$$.lvec.addAll($3.lvec);

$$.evec.addAll($3.evec);

}

}

| /\* empty \*/

;

**1.2.4 取子数组和数组下标动态访问**

数组下标静态访问、取子数组、数组下标动态访问三者有公因子Expr[Expr，数组下标静/动态访问有公因子Expr[Expr]，按如下方式消除左公因子：

ExprT8 : '[' Expr ExprT9

{

SemValue sem = new SemValue();

if ($3.counter == Tree.COLON)

{

sem.expr = new Tree.Range($2.expr, $3.expr, $3.loc);

}

else if ($3.counter == Tree.DEFAULT)

{

sem.expr = new Tree.Default($2.expr, $3.expr, $3.loc);

}

else

{

sem.expr = $2.expr;

}

$$.vec = new Vector<SemValue>();

$$.vec.add(sem);

if ($3.vec != null) {

$$.vec.addAll($3.vec);

}

}

| '.' IDENTIFIER AfterIdentExpr ExprT8

{

SemValue sem = new SemValue();

sem.ident = $2.ident;

sem.loc = $2.loc;

sem.elist = $3.elist;

$$.vec = new Vector<SemValue>();

$$.vec.add(sem);

if ($4.vec != null) {

$$.vec.addAll($4.vec);

}

}

| /\* empty \*/

;

ExprT9 : ']' ExprT10

{

$$.loc = $2.loc;

$$.counter = $2.counter;

if ($$.counter == Tree.DEFAULT)

{

$$.expr = $2.expr;

}

else

{

$$.vec = new Vector<SemValue>();

if ($2.vec != null) {

$$.vec.addAll($2.vec);

}

}

}

| ':' Expr ']'

{

$$.loc = $1.loc;

$$.counter = Tree.COLON;

$$.expr = $2.expr;

}

;

ExprT10 : ExprT8

{

$$.vec = new Vector<SemValue>();

if ($1.vec != null) {

$$.vec.addAll($1.vec);

}

}

| DEFAULT Expr9

{

$$.loc = $1.loc;

$$.counter = Tree.DEFAULT;

$$.expr = $2.expr;

}

;

**\*1.2.5 变量标识符后的逗号问题**

新特性并不涉及变量标识符后的逗号，但是在错误恢复时对变量标识符后逗号的处理会影响报错的个数。具体来说，在VariableDef中，如下的表达式是不允许的，

int x, y;

但是由于Variable可以出现在函数参数列表里，所以逗号在Variable的预测集里，这样int x, y; 的错误就会被parser程序识别并跳过Variable非终结符，引起后续分析的错误。为此，将函数列表中的Variable定义为FuncVariable加以区分。

VariableList : FuncVariable SubVariableList

{

$$.vlist = new ArrayList<VarDef>();

$$.vlist.add($1.vdef);

if ($2.vlist != null) {

$$.vlist.addAll($2.vlist);

}

}

;

SubVariableList : ',' FuncVariable SubVariableList

{

$$.vlist = new ArrayList<VarDef>();

$$.vlist.add($2.vdef);

if ($3.vlist != null) {

$$.vlist.addAll($3.vlist);

}

}

| /\* empty \*/

;

FuncVariable : Type IDENTIFIER

{

$$.vdef = new Tree.VarDef($2.ident, $1.type, $2.loc);

}

;

**2. 空else分支冲突处理**

PS(ElseClause->Else Stmt) ∩ PS(ElseClause->/\* empty \*/) = { ELSE }，因此不符合LL(1)文法。解决方法是规定ElseClause->Else Stmt的优先级高于ElseClause->/\* empty \*/，实现方法是从PS(ElseClause->/\* empty \*/)中去掉ELSE，使得(if, else)对总是按就近匹配的原则进行匹配。

例如decaf程序段

if (a>1)

if (a>2)

a = 3;

else

a = -3;

如果没有规定优先级，则从

IF '(' Expr ')' Stmt ElseClause

中令ElseClause->/\* empty \*/，则可以推导出

if (a>2)

a = 3;

令ElseClause->ELSE Stmt，则可以推导出

if (a>2)

a = 3;

else

a = -3;

但是，由于pg.jar中引入了优先级机制，ElseClause在遇到ELSE时总是先执行ELSE Stmt的推导，因此else语句与第二个if语句匹配，第一个if语句的ElseClause则导出/\* empty \*/.

**3. 数组comprehension表达式用[]表示为什么改写成LL(1)文法比较困难？**

Expr8 : Expr9 ExprT8

{

$$.expr = $1.expr;

$$.loc = $1.loc;

if ($2.vec != null) {

for (SemValue v : $2.vec) {

if (v.expr != null) {

$$.expr = new Tree.Indexed($$.expr, v.expr, $$.loc);

} else if (v.elist != null) {

$$.expr = new Tree.CallExpr($$.expr, v.ident, v.elist, v.loc);

$$.loc = v.loc;

} else {

$$.expr = new Tree.Ident($$.expr, v.ident, v.loc);

$$.loc = v.loc;

}

}

}

}

| ‘[‘ Expr FOR IDENTIFIER IN Expr IfCompreClause

{

$$.expr = new Tree.Comprehension($2.expr, $4.ident, $6.expr, $7.expr, $1.loc);

}

;

PS(Expr8->‘[‘ Expr FOR IDENTIFIER IN Expr IfCompreClause) = {‘[‘}

且

{‘[‘}∈PS(Expr8->Expr9 ExprT8)

故

PS(Expr8->Expr9 ExprT8) ∩ PS(Expr8->‘[‘ Expr FOR IDENTIFIER IN Expr IfCompreClause) = {‘[‘}

不符合LL(1)文法，按照pg.jar的优先级原则，要从PS(Expr8->‘[‘ Expr FOR IDENTIFIER IN Expr IfCompreClause)去掉’[‘，这样

PS(Expr8->‘[‘ Expr FOR IDENTIFIER IN Expr IfCompreClause) = ∅

即永远无法推导出数组comprehension表达式。

**4. 语法错误误报的例子**

class Main {

static void main() {

int x[], y;

}

static void main() {

int x = 1;

}

}

如上代码，总共有3处错误：

（1）第3行变量x后面不能加[]，定义数组方式错误

（2）第3行一个变量定义中不能用逗号分隔一次定义两个变量

（3）第7行不能在定义变量的同时为变量赋初值

因此，文法分析程序期望的输出为

\*\*\* Error at (3,14): syntax error

\*\*\* Error at (3,16): syntax error

\*\*\* Error at (7,9): syntax error

但实际输出为

\*\*\* Error at (3,14): syntax error

\*\*\* Error at (3,16): syntax error

第3个错误没有被发现。从语法分析的过程查找原因：

（1）在第3行“int x[”处解析非终结符Variable时，期望遇到终结符’;’作为结束，却遇到终结符’[‘，报错“\*\*\* Error at (3,14): syntax error”

（2）’[]’被按照数组常量解析，解析完后非终结符A变为Stmt，终结符lookahead变为 ’,’，’,’不在A的预测集里，报错“\*\*\* Error at (3,16): syntax error”. 但是’,’在follow集合中，故parse函数跳过非终结符Stmt；非终结符A变为FieldList，lookahead仍为’,’，此时’,’仍在follow集合中，因此跳过FieldList；如此不断跳过各个非终结符，lookahead却始终为’,’，直到decaf.jar认为程序已经分析完成。由此可见，’,’以后的程序不会被分析到，之后的语法错误也不会被报出。