

麻省理工学院

电气工程与计算机科学

6.035, 2005 年秋

分发资料 8——语义分析项目

9 月 21 日, 星期三

完成时间: 10 月 6 日, 星期四

警告: 项目的静态语义部分需要较之前部分更多的努力。

学生需要将编译器的功能扩展到找到, 报告和恢复 Decaf 程序语言中的语义错误。大多数通过测试规则可以检测出的语义错误都在分发资料 6 中的“语义规则”中列示出。在本分发资料结尾处, 对规则也进行了重复。但是, 学生也应该认真阅读分发资料 6 中得详尽描述, 以确保编译器能够捕捉到语言定义得所有语义错误。我们也试图提供语义规则得精确表述, 但是如果学生仍然觉得语言定义得语义规则可能有多种理解, 学生可以就自认为最合理得理解进行工作, 并在项目文档中清楚列举你的假设。

这部分项目包括以下工作:

1. 在 CUP 语法分析规范中添加语义行为, 构建一个高级中间表达式树, 并且将所选择的语义检测作为语法检测处理的部分内容来执行。如何构建中间表达的问题在讲座中会进行讨论, 在这个分发资料末节也会有一些相关暗示。

当驱动在调试模式下运行时, 驱动要以适合调试且易于阅读的形式, 很好的打印已构建的中间表达式树。

2. 为类构建符号表。(一张符号表就是一个运行环境, 如从标志符到类似于标量声明的语义目标。环境是分级架构的, 和源级架构如类体, 函数体, 循环体等。)

3. 通过中间表达执行所有剩余的语义检测, 并访问符号表。

注意: 这部分作业不需要运行时间检测。

提交

在书写项目的时候, 请遵守在分发资料 3 中规则。学生的设计文档需要包括对如何组建中间表达和符号表结构的描述, 另外还需要包括如果执行语义检测的讨论。

提交的电子部分和之前阶段相似。在小组文件夹上提交一个名为 `leNN-semantic.tar.gz` 的压缩 tar 文件, NN 为学生所在小组名。文件必须包含所有相关源文件和一个 `makefile`。另外, 学生还要提供一个由 Java 工具生成 Java 文档, 在相同目录下命名为 `leNN-semantic.jar`。解压 tar 文件在运行 `make` 应该可以生成同样的 Java 文档。With the `CLASSPATH` set to `leNN-parser.jar:<directory>/java_cup.jar:...`, 学生的编译器应该能够从以下命令行开始运行:

```
java Compiler<文件名>
```

输出的最终结果应该是一个在编译文件过程中遇到的所有错误的报告。编译器要能够对检测到的所有错误汇报合理且准确的错误消息 (要有行号和标志符名)。要避免对同一错误进行多次报错。举例来说, 如果在表达式 “`x=y+1`” 中, `y` 没有声明, 编译器应该仅仅对 `y` 进行一次报错, 而不是对 `y` 进行报错, 在对 `+` 进行报错, 还对整个表达式进行报错。

在实现语义检测器后, 编译器要能够检测并报告所有 Decaf 输入程序的静态错误 (如编

译时间错误), 包括在之前部分检测到的词法和语法错误。另外, 对合法的 Decaf 程序, 编译器不能报告任何信息。但是, 我们不期望学生避免由错误恢复造成的伪错误信息。编译器在一些情况下报告伪语义错误是可能的, 这取决于编译器句法错误恢复的有效性。

词法检测器要能够检测输入的十进制和十六进制整数(词法分析器只是把这些整数作为字符串存储)的数值, 这些数值的范围在分发资料 5 中有详细说明。

如果之前提到的一样, 编译器要有调试模式, 在这种模式下, 编译器构建的中间表达和符号表数据结构可以某种形式打印。可从下面命令行执行:

```
java Compiler debug <文件名>
```

测试

课程服务器在提供了测试案例。

请阅读测试案例的注释, 确认我们希望对编译器的要求。分数就是根据编译器在这些案例和一些隐藏测试中的表现确定的。为了得到满分, 完整的文档也是必需的。

语义规则

这些规则除了在语法中的限制外, 增加了合法 Decaf 语言的限制。一个程序, 如果没有任何语法错误同时也没有违反下面的任何规则, 就称为合法程序。功能强大的编译器要能就下面每条规则进行检测, 并且对每条非法程序产生至少一条错误消息, 但对合法程序不进行报错。

1. 在同一 scope 内, 不对标志符重复声明两次。
2. 标志符在被声明之前不能使用。
3. 程序包含一个无参数的 main method (注意, 由于程序从 main 函数开始执行, 所以在 main 之后定义的函数都不会执行)。
4. 在数组申明的 `<int_literal>` 都必须大于 0。
5. 函数调用中的数据和类型必须和形参的数据和类型相同, 如签名必须完全一样。
6. 如果函数调用是以表达式形式出现的, 则函数必须返回值。
7. Return 函数不一定要有返回值, 除非它是在声明需要返回值的函数体中。
8. Return 声明中的表达式必须函数定义嵌套内层的声明返回值相同。
9. 用为 `<location>` 的 `<id>` 必须对一个局部/全局变量或者形参命名。
10. 对于 `<id>[<expr>]` 的区域
 - (a) `<id>` 必须为数组变量, 并且
 - (b) `<expr>` 的类型必须为 `int`。
11. If 和 while 中的 `<expr>` 类型必须为布尔型。
12. `<airth-op>` 和 `<rel-op>` 的操作数必须为 `int` 型。
13. `<eq-op>` 的操作数必须为同一类型, 要么是 `int` 型, 要么为布尔型。
14. `<cond-op>` 和逻辑非 (!) 的操作数必须为布尔型。
15. `<location>` 和 `<expr>` 在 `<location> = <expr>` 中必须有同一类型。

16. Break 和 continue 必须包含在循环体中。

实现建议

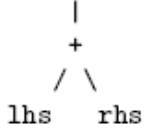
• 学生需要在中间表达中对每个网点声明类。在很多地方，中间表达网点的层次结构和语言语法相类似。举例来说，层次树的部分可能看起来像（缩进代表着层次结构）：

```
abstract class Ir
abstract class      IrExpression
abstract class      IrLiteral
    class            IrIntLiteral
    class            IrBooleanLiteral
    class            IrCallExpr
    class            IrMethodCallExpr
    class            IrCalloutExpr
    class            IrBinopExpr
abstract class      IrStatement
    class            IrAssignStmt
    class            IrBreakStmt
    class            IrContinueStmt
    class            IrIfStmt
    class            IrWhileStmt
    class            IrReturnStmt
    class            IrInvokeStmt
    class            IrBlock
    class            IrClassDecl
abstract class      IrMemberDecl
    class            IrMethodDecl
    class            IrFieldDecl
    .
    .
    .
    class            IrVarDecl
    class            IrType
```

如前所述的类实现了输出程序抽象语法树。在其最简单形式中，每个类就是其子树的一个原组。举例如下：

```
public class IrBinopExpr extends IrExpression
{
    private final int      operator;
    private final IrExpression lhs;
    private final IrExpression rhs;
}

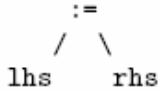
```



or:

```
public class IrAssignStmt extends IrStatement
{
    private final IrLocation  lhs;
    private final IrExpression rhs;
}

```



另外，学生需要对程序的寓义实体定义类，以表示其抽象属性（如表达类型，方法签名，类描述符等等。）和建立他们的对应关系。如：每个表达都有一种类型；每个变量声明都引入了一个变量；每个块都定义了一个范围。很多这些属性都源于树的递归二叉树的遍历。学生应该尽量多的对符号标准类进行举例，这样可以与使用同等的参数形成对比。因为学生会在以后几个月中和这些类相伴，所以强烈建议学生仔细设计这些类，使用优秀的软件工程并形成文档。

- 所有的错误信息都需要包含文件名和与错误信息最相关符号所在的行号（请使用你自己的判断）。这就意味着，在构建抽象语法树（AST）时，必须确认每个中间表达网们都包含足够的信息能在稍后时间确定行号。

（一些工业编译器，如 Java 的 Jlikes，构架中以 start 作为开始字符，以 end 作为结束字符的每个中间表达网点，都允许错误信息准确报告源文件中哪一行包含了错误。JavaCUP 页可以轻易完成这项功能，它可以显示出在变量 left 和 right 间的每个语法符号的字符位置。不应该在遇到输入错误时就发生异常：这样的话可能导致每运行一次编译器都至多能报告一个错误。优秀的前端可以通过在停止之前报告多个错误来节省用户时间，同时允许程序员在不得不重启编译器前对一些错误进行修改。

- 语义检测应该是自顶向下的。LR 语法分析器中重新调用 AST 则是自低向上的。尽管语义检测中的类型检测可以以自底向上的方式进行，其它的检查类型（如检测未定义变量的使用）则不行。

完成这部分有两种方法。第一种方法是利用生成中间过程中的语法分析器行为，如：

```
block ::= LBRACK                                {: envs.push(new Env()); :}
        statements:s                            {: checkStmts(s); :}
        RBRACK                                  {: envs.pop(); :}

```

在这个生成过程中，创建了新的环境并压入环境栈，在环境栈的上下文中检测块的主体，在达到块的末端后丢弃新的环境。这种实现方法所需代码较少呆更为复杂，因为它改变了语法的生成规则，可能导致冲突。

另外一种更高明的方法是在语法分析完成后，在完整的 AST 上调用语法检测器。在这种方法种的块伪代码类似于：

```
void checkBlock(EnvStack envs, Block b) {
    envs.push(new Env());
    foreach s in b.statements
        checkStatement(envs, s);
    envs.pop();
}
```

因而，在此语法检测可以视为 AST 的一个访问者（学生可从课程 6.170 或设计模板上熟悉）。（从代码生成开始，某些优化和良好的打印都可自然的视做访客，我们推荐这种方法作为更高明实现的方法）

- 对于负整数需要一些关注。从之前的分发资料看，负整数事实上是两个分离的符号：正整数和前面的‘-’。任何时候，自顶向下语义检测器发现一个一元负算子，监测器应该检测其操作数是否是一个正整数，如果是，则用单个负数取代子结构树（两个网点都需要取代）。