

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 6——Decaf 语言

9 月 7 日, 星期三

课程的项目是用称为 Decaf 的语言编写一个编译器。Decaf 是一种和 C 或 Pascal 类似的强制性语言。

词汇说明

所有的 Decaf 关键字都是小写的。关键字和标志符 case-sensitive. 比如, if 就是一个关键字, 而 IF 则是一个变量名; foo 和 Foo 则是两个截然不同变量名。

保留字为

boolean break callout class continue else false if int return true void while

注意, **Program** (见下) 不是一个关键字, 而是一个在特定环境下使用有特殊意义的标志符。

注释以//开始, 在这一行末结束。

词法记号间可能出现空白。空白的定义是一个或多个空格, 制表符, 页、断行符, 和注释。

关键字和标志符间必须用空白, 或既不是关键字也不是标志符的符号隔开。比如, **thiswhiletrue** 是一个标志符, 而不是三个不同的关键字。如果一个字符串, 以字符或下划线开头, 则连同其后所跟的字符串形成一个 token.

字符串文本由双引号内的字符组成。字符串文本包含单引号内的字符型数据。

Decaf 中的数字为带符号 32 位的, 也就是十进制-2147483648 到 2147483647 之间。如果一个序列以 Ox 开头, 则这两个字符以及其后的[0-9a-fA-F]形成了十六进制整数字符。如果一个序列以一个十进制书开头 (但不是 Ox), 那么这个十进制数长前缀就为十进制整数字符。注意, 稍后会进行边界检测。以长串数字 (如 123456789123456789) 在词法分析看来仍然是单个 token.

一个<char>类型可以为任何可打印的 ASCII 字符(ASCII 值为十进制数 32 到 126 之间, 或者八进制数 40 到 176 之间), 除了引号 (“), 单引号 (‘), 或反斜线符号 (\), 加上 2 个 “\” 表示引用, “\” 表示单个引用, “\\” 表示反斜线符号, “\t” 表示制表符, “\n” 表示换行。

相关语法

含注释

<foo>	意为 foo 为非终结符
foo	(粗体) 意为 foo 为终结符 如: 一个 token 或 token 的一部分
[x]	意为 x 的 0 个或 1 个匹配项, 如, x 为可选的; 注意引号中的括弧[‘’]是终结符
x*	x 的 0 个或多个匹配项.
x+,	一个或多个 x 的逗号分隔清单

{}	大括弧用于分组； 注意引号中的括弧‘{}’为终结符
	隔离二选一选项

$\langle \text{program} \rangle \rightarrow \text{class Program } \{ \langle \text{field_decl} \rangle^* \langle \text{method_decl} \rangle^* \}$
 $\langle \text{field_decl} \rangle \rightarrow \langle \text{type} \rangle \{ \langle \text{id} \rangle \mid \langle \text{id} \rangle \text{ '[' } \langle \text{int_literal} \rangle \text{ ']' } \}^+, ;$
 $\langle \text{method_decl} \rangle \rightarrow \{ \langle \text{type} \rangle \mid \text{void} \} \langle \text{id} \rangle ([\{ \langle \text{type} \rangle \langle \text{id} \rangle \}^+,]) \langle \text{block} \rangle$
 $\langle \text{block} \rangle \rightarrow \{ \langle \text{var_decl} \rangle^* \langle \text{statement} \rangle^* \}$
 $\langle \text{var_decl} \rangle \rightarrow \langle \text{type} \rangle \langle \text{id} \rangle^+, ;$
 $\langle \text{type} \rangle \rightarrow \text{int} \mid \text{boolean}$
 $\langle \text{statement} \rangle \rightarrow \langle \text{location} \rangle = \langle \text{expr} \rangle ;$
 $\mid \langle \text{method_call} \rangle ;$
 $\mid \text{if} (\langle \text{expr} \rangle) \langle \text{block} \rangle [\text{else } \langle \text{block} \rangle]$
 $\mid \text{while} (\langle \text{expr} \rangle) \langle \text{block} \rangle$
 $\mid \text{return} [\langle \text{expr} \rangle] ;$
 $\mid \text{break} ;$
 $\mid \text{continue} ;$
 $\mid \langle \text{block} \rangle$
 $\langle \text{method_call} \rangle \rightarrow \langle \text{method_name} \rangle ([\langle \text{expr} \rangle^+,])$
 $\mid \text{callout} (\langle \text{string_literal} \rangle [, \langle \text{callout_arg} \rangle^+,])$
 $\langle \text{method_name} \rangle \rightarrow \langle \text{id} \rangle$
 $\langle \text{location} \rangle \rightarrow \langle \text{id} \rangle$
 $\mid \langle \text{id} \rangle \text{ '[' } \langle \text{expr} \rangle \text{ ']'}$

```

    <expr> → <location>
           | <method_call>
           | <literal>
           | <expr> <bin_op> <expr>
           | - <expr>
           | ! <expr>
           | ( <expr> )

    <callout_arg> → <expr> | <string_literal>

    <bin_op> → <arith_op> | <rel_op> | <eq_op> | <cond_op>

    <arith_op> → + | - | * | / | % | << | >> | >>>

    <rel_op> → < | > | <= | >=

    <eq_op> → == | !=

    <cond_op> → && | ||

    <literal> → <int_literal> | <char_literal> | <bool_literal>

    <id> → <alpha> <alpha_num>*

    <alpha_num> → <alpha> | <digit>

    <alpha> → a | b | ... | z | A | B | ... | Z | _ | .

    <digit> → 0 | 1 | 2 | ... | 9

    <hex_digit> → <digit> | a | b | c | d | e | f | A | B | C | D | E | F

    <int_literal> → <decimal_literal> | <hex_literal>

    <decimal_literal> → <digit> <digit>*

    <hex_literal> → 0x <hex_digit> <hex_digit>*

    <bool_literal> → true | false

    <char_literal> → ' <char> '

    <string_literal> → " <char>* "

```

语义

Decaf 程序包含名为 **Program** 的单个类声明。类声明包含了域声明和函数声明。域声明表明了可以被程序中所有的函数调用的标量。函数声明表明了函数。程序必须包含一个没有参数的 **main** 函数。程序从 **main** 函数开始执行。

类型

Decaf 语言中有两种基本类型—**整数型**和**布尔型**。另外，还有整数数组（int[N]）和布尔数组（boolean[N]）。

数组可以在全局范围内声明（**class declaration**）。所有的数组都是一维的，并有与编译时间相符的大小。数组的变址为 0 到 N-1，N>0 为数组的大小。一般括弧符号用来标注数组。由于数组有与编译时间相符的大小并且不能作为参数被声明（或者作为本地变量），所以对于查询 Decaf 语言中数组变量的长度是比较困难的。

范围规则

Decaf 语言的范围规则简单且有限。所有的标志符都必须在使用前定义（原文的）。例如：

- 变量必须在使用前定义。
- 函数只能被其头函数之前的代码调用。（注意递归函数是可以的）

在 Decaf 程序中，至少都有两种合法的范围：全局范围和函数范围。全局范围包含了域名和 **Program** 类声明引入的函数。函数范围包含了变量名和函数声明所引入的形参。另外的局部范围存在于代码中的每个块中；可接在 **if** 和 **while** 后，或者插入在任何表达式合法的位置。函数范围内的标志符可以映射全局范围内的标志符。同样，局部范围内的标志符可以映射浅一层嵌套范围、函数范围和全局范围中的标志符。

函数或局部范围内定义的变量名可能会映射全局范围内的函数名。在这样的情况下，在变量离开范围之前，标志符都只能作为变量使用。

在同一个范围内，标志符定义次数不能多余一次。**field** 和函数名必须在局部范围内区分开，局部变量名和形参在一个局部范围内必须区分开。

有效区域

Decaf 语言有两种范围：局部/全局变量和（全局）数组元素。每个范围都有一个类型。int[N]和 boolean[N]型 **location** 表示数组元素。由于 Decaf 语言中的数组大小为静态的，所以数组要定位在程序的静态数据空间中并且不能定位在堆中。

在每个范围声明时都被初始化为一个默认值。整数默认值为零，布尔数默认值为假。局部变量在进入声明范围时必须初始化。在程序开始时必须堆数组元素进行初始化。

赋值

赋值只允许用数量值。对于 int 和布尔型，Decaf 使用值复制语义，赋值语句 <location>=<expr>则是把<expr>中的计算值拷贝到<location>中。对于数组类型，<location>和<expr>必须要有为数字值单个数组参数。

<location><expr>必须为同样的类型。

在一个函数体内对形参变量赋值是合法的。这样的赋值旨在函数范围内起作用。

函数调用和返回

函数调用涉及到（1）把参数值从调用者传递到被调用者，（2）执行被调用者，和（3）返回调用者，可能会有返回值。

参数传递通过赋值传递：函数的形参当作函数本地变量来考虑，并且函数将其初始化为参数表达的赋值。参数自左向右进行赋值。

这样，通过顺序执行函数的表达式，被调用者也就被执行了。

函数如果没有声明结果类型，就只能叫做基本指令，如，它就不能用作表达式。在 **return** 指令调用时（不允许结果表达式），或达到被调用者程序末端时，函数将控制权返回给调用者。

当基本指令执行时，如果调用值就是 **return** 指令的计算值，这样返回值的函数可称为表达式的一部分。控制者不能执行到要返回值的函数文本末端，这样是不合法的。

返回值的函数也可称为基本指令。在这种情况下，忽略返回值。

控制指令

if 指令语义和其它程序语言相似。首先，计算表达式。如果结果为真，则执行真的分支。否则，如果 **else** 分支存在的话，执行 **else** 分支。由于 Decaf 语言要求 **true** 和 **else** 分支要圈在括弧中，所以在 **else** 和与之相对的 **if** 语句配对方面是很清晰的。

if 指令语义和其它程序语言相似。首先，计算表达式。如果结果为假，则跳出循环。否则，执行循环体。如果控制达到了循环体末端，则再次执行 **while** 语句。

相似的，**break** 和 **continue** 指令语义也和其它程序语言相似的含义：**break** 退出最内层循环，继续机型下一个语句；**continue** 则跳转到内层循环体开始位置并重新计算条件。

表达式

表达式遵守一般的计算规则。在没有其它限制条件下，同一段程序的算子从左到右执行。要优于一般规则，可加表达式外园括弧。

局部表达式计算出的值包含在这个局部范围内。

函数调用表达式在函数调用和返回讨论。数组算子在类型进行讨论。I/O 比较表达式在库编号进行讨论。

整型计算结果为其整数值。字符型计算结果为其整数 ASCII 值，如，'A' 为整数 65。（字符类型值为 **int**。）

算术算子（**<arith_op>**和负一元数）在处理比较算子（**<rel_op>**）都有其一般优先级和含义。% 计算算子进行除法运算后的余数。<<和>>分别为结构左右移。>>>为逻辑右移。

比较算子还可以进行比较计算。等式，==和!=只定义为 **int** 和 **boolean** 型，这两个算子可以对同种类型的任意两个表达式进行比较。（==是相等，!=是不等）。

比较算子和等式算子的结果类型为布尔型。

算子优先级，从高到低：

算子	注释
-	负一元数
!	逻辑非
*/%	乘号，除号，求余
+-	加，减
<<>>>>	移位
<<=>=>	比较
== !=	等式
&&	条件与

	条件或
--	-----

注意，这些规定不在相关语法中反映。

库编号

Decaf 语言包含了调用函数的简单方法，提供了运行时间系统，如标准 C 语言库或用户定义函数。

简单调用函数方法是：

`int callout (<string_literal>,[<callout_arg>+,])`—该函数由初始字符串命名，调用该函数，同时提供的参数也传递到函数中。布尔和整数型表达都已整数型表达传递；字符串和数组型表达式以指针传递。函数的返回值以整数型返回。`callout` 函数的调用者需要保证参数和函数特性相符，并且保证如果基本库函数返回适当类型值后，仅仅使用返回值。参数是传递到系统标准调用惯例函数的。

为了能使用 `callout` 访问标准 C 语言库，I/O 口函数可以以 C 语言编写（或者其他的任何语言），用标准工具进行编译，用运行时间系统链接，最后用 `callout` 访问。

语义规则

这些规则是在语法限制之外附加的正确 Decaf 程序语言的限制。语法上是合适的并且不违背下面的任何条款的程序就称为合法程序。一个强健的编译器可以准确的检测每一条规则，并且可以生成错误信息，描述它可以找到的违反规则行为。一个强健的编译器可以对每个非法程序生成至少一个错误消息，但是不能对合法程序生成错误消息。

1. 在同一个范围内不那果农对一个标志符重复定义两次。
2. 标志符在使用之前必须先声明。
3. 程序包含一个称为 `main` 函数的定义，`main` 函数没有参数（注意由于程序从 `main` 函数开始执行，任何定义在 `main` 函数之后的函数时不会被执行的）。
4. 数组声明中的 `<int_literal>` 必须要大于 0。
5. 函数调用参数的数值和类型必须和形参的数值和类型一致：如特点必须完全相同。
6. 如果函数作为一个表达式进行调用，则函数必须要返回值。
7. `return` 指令不可返回值，除非它出现在声明要返回值的函数体中。
8. `return` 命令中的表达式的返回值必须要和内层函数定义声明的返回值类型相同。
9. 用作 `<location>` 的 `<id>` 必须命名一个声明了的局部/全局变量，或者是形参。
10. 对于所有 `<id>[<expr>]` 形式的 `location`
 - (a) `<id>` 必须要为数组变量，并且
 - (b) `<expr>` 类型必须为 `int`。
11. `if` 和 `while` 命令中的 `<expr>` 类型必须为布尔型。
12. `<airth_op>` 和 `<rel_op>` 的操作数必须要有 `int` 类型的。
13. `<eq_op>` 操作数必须类型一直，或者是 `int` 或布尔型。
14. `<cond_op>` 和逻辑非 (!) 的操作数必须为布尔型。
15. 在一个 `assignment` 中的 `<location>` 和 `<expr>`，`<location>=<expr>`，必须类型相同。
16. 所有的 `break` 和 `continue` 命令必须限制在一个循环体内。

运行时间检测

上述的限制是由编译器语义检测器静态强制要求的限制，除了这些限制以外，下面的限制是动态强制要求的：编译器的代码生成器必须要能生成执行这些检测的代码；在运行时间检测时要发现这些违反规则行为。

- 1.数组的下表必须不能越界。

- 2.即时到了声明要返回值的函数在末端，对函数的控制也不能减弱。

在运行时间错误发生时，要能输出到终端合适的错误信息并且停止程序执行。这样的错误信息有利于程序员找出源程序的问题。