Decaf Language Definition

Anoop Sarkar – anoop@cs.sfu.ca

September 10, 2007

1 Introduction

The programming assignments over the semester will build various components towards a working compiler for a programming language called **Decaf**. This document is the specification for the **Decaf** language[1]. This language definition might evolve over the course of the semester. Each version of the language definition will be marked with the date. Please refer to the latest version available.

Decaf is a strongly typed, object-oriented language with support for inheritance and encapsulation. The design of **Decaf** has many similarities with other programming languages that are familiar to us such as *C, C++* or *Java*. Although, keep in mind that **Decaf** is not an exact match to any of those languages and has its own peculiar properties. The feature set has been trimmed down considerably from what is usually part of a full-fledged programming language. This was done to keep your programming assignments manageable. Despite these limitations, the **Decaf** compiler will be able to handle interesting and non-trivial programs.

2 Sample Decaf Code

Here is a simple program written in the **Decaf** language:

```
class GreatestCommonDivisor {

    int a = 10;
    int b = 20;

    void main() {
       int x, y, z;
       x = a;
       y = b;
       z = gcd(x, y);

       // print_int is part of the standard input-output library
       callout("print_int", z);
    }

    // function that computes the greatest common divisor
    int gcd(int a, int b) {
       if (b == 0) { return(a); }
       else { return( gcd(b, a % b) ); }
    }

}
```

[1] Like Java, but with less caffeine.

3 Notation

⟨foo⟩	means ⟨foo⟩ is a non-terminal symbol
foo	means **foo** is a terminal symbol, i.e. a token recognized by the lexical analyzer
‘;’	indicates a terminal/token that is either an operator like **‘<=’** or a single char punctuation like **‘;’**
$\left[\text{x}\right]$	means zero or one occurrence of x, i.e. x is optional n.b.: do not confuse this notation with terminals **‘[’** and **‘]’**
x^*	zero or more occurrences of x
x^+	one or more occurrences of x
$\left\{ \quad \right\}$	curly braces are used for grouping items n.b.: do not confuse this notation with terminals ‘{’ and ‘}’
$\left\{ \text{x} \right\}^+$**,**	a comma-separated list of one or more x’s e.g. `i` or `i,j,k`
|	separates alternatives for lhs of a CFG rule or a group $\left\{ \quad \right\}$ n.b.: the context determines whether | separates CFG rules e.g. $\left\{ \langle \text{type} \rangle \mid \textbf{void} \right\}$ vs. $S \rightarrow aS \mid a$
`[char1-char2]`	denotes a group of characters (for token definitions) e.g. `[a-c]` denotes the three characters `a, b, c`

4 Lexical Considerations

All **Decaf** keywords are lowercase. Keywords and identifiers are case-sensitive. For example, **if** is a keyword, but `IF` is an identifier. Also, `foo` and `Foo` are two distinct identifiers.

Comments are started by **//** and are terminated by the end of the line.

4.1 Token Definitions

The keywords are:

bool break callout continue class else extends false
for if int new null return rot true void while

For now the keywords **extends** , **new** and **null** will not appear elsewhere in the language definition; these keywords are reserved for ”future extension”.

The operator and punctuation tokens are:

‘{’ ‘}’ ‘[’ ‘]’ ‘,’ ‘;’ ‘(’ ‘)’ ‘=’ ‘-’
‘!’ ‘+’ ‘*’ ‘/’ ‘<<’ ‘>>’ ‘<’ ‘>’ ‘%’
‘<=’ ‘>=’ ‘==’ ‘!=’ ‘&&’ ‘||’ ‘.’

The **‘.’** token is reserved for ”future extension”.

Tokens such as **stringConstant** which defines a string constant like `"hello, world"`; or single character tokens, such as, **‘;’** or **‘.’** do not appear in the list of keywords above but are valid tokens and are used in defining the reference grammar of **Decaf** (see Section 5).

Identifiers, denoted by the token **id** are defined as starting with an alphabetic character or the underscore character `[a-zA-Z_]` and followed by zero or more alphanumeric characters including the underscore character `[a-zA-Z_0-9]`.

Keywords and identifiers must be separated by white space, or a token that is neither a keyword or an identifier. **thiswhiletrue** is a single identifier, not three distinct keywords. See Section 4.2 for some examples.

String constants, denoted by the token **stringConstant** will have a lexeme value that is composed of characters enclosed in double quotes. A string must start and end on a single line, it cannot be split over multiple lines. Note that the string constant can include escape sequences like \n and this is distinct from a newline character inside the string constant. For example, "\n" is legal, but "
" is not legal. For more on strings and character tokens see Section 4.3.

Integer constants in **Decaf**, denoted by the token **intConstant** , are either decimals (base 10), or they are hexadecimals (base 16). A hex integer constant must begin with `0x` (that's a zero, not the letter 'o') and followed by a sequence of hex digits which include with the decimal digits plus the letters `a` through `f` (either upper or lowercase). Examples of integer constants: `8, 012, 0x0, 0x12aE`

Character constants, denoted by the token **charConstant** will have a lexeme value that is a single character enclosed in single quotes. A character constant is any printable ASCII character (ASCII values between decimal value 32 and 126, or octal 40 and 176) but a character constant *cannot be* a single quote (''), or backslash ('\') character. For more on strings and character tokens see Section 4.3. Character constants also include the character sequences, e.g. '"' to denote a quote, '\'' to denote a single quote, '\\' to denote backslash. All the escape sequences are listed in Section 4.4.

You can choose to include special tokens for whitespace and comments. You can assume some intermediate step before parsing occurs can strip out these special tokens. This makes the lexical analyzer more consistent since a special definition is used for both whitespace and comments, and also makes the definition of string constants easier.

4.2 Token Boundaries and Whitespace

The boundaries between tokens such as integer constants, keywords and identifiers are explained using the following rules. In effect, these rules define an algorithm for breaking up a sequence of characters from the set `[0-9a-zA-Z_]` into tokens.

- If the sequence begins with `0x` then these first two characters, and the longest subsequence of characters immediately following them drawn from the set `[0-9a-fA-F]` form a hex integer constant. The last such character is the end of the token.
- If the sequence begins with a decimal digit (but not `0x`) then the longest prefix of decimal digits forms a decimal integer constant. The last such character is the end of the token. Note that the semantics of range checking occurs later, so that a long sequence of digits, e.g. `123456789123456789` which is clearly out of range is still scanned as a single token. The semantic analyzer will come in later and reject this lexeme value as a valid integer constant.
- If the sequence begins with an alphabetic character or _ then this character and the longest sequence of alphanumeric characters `[0-9a-zA-Z_]` following this initial character forms a token which is either an identifier or a keyword.
- **Whitespace** and other token definitions can play a role in identifying token boundaries, e.g. the string `rot3` is a single identifier token but `rot 3` is two tokens, one a keyword **rot** and the second an integer token **3**, and `rot(3)` can only be a sequence of 4 tokens, the keyword **rot**, the left parenthesis, the integer token **3** and the right parenthesis[2]. Whitespace is defined as one or more of the characters *\t, \n, \v, \f, \r* plus the ASCII space character.

Here are some more examples that explain these rules:

```
0x123food  = INTCONST(0x123f), IDENTIFIER(ood)
0xfood123  = INTCONST(0xf), IDENTIFIER(ood123)
```

[2]Note that this example is here only to explain how things work in lexical analysis. The actual syntax for using **rot** is `0x001 rot -2` or `0x001 rot 2` for right/left rotation respectively. See Section 6.2 for more details on syntax of expressions.

```
123break   = INTCONST(123), KW_BREAK
0x123rot3  = INTCONST(0x123), IDENTIFIER(rot3)
0x123rot 3 = INTCONST(0x123), KW_ROT, INT(3)
1250x356   = INTCONST(1250), IDENTIFIER(x356)
break123   = IDENTIFIER(break123)
breakwhile = IDENTIFIER(breakwhile)
```

4.3 String and Character Constants

String constants, denoted by the token **stringConstant** will have a lexeme value that is composed of characters enclosed in double quotes. A string must start and end on a single line, it cannot be split over multiple lines. Note that the string constant can include escape sequences like \n and this is distinct from a newline character inside the string constant. For example, "\n" is legal, but "
" is not legal.

Character constants, denoted by the token **charConstant** will have a lexeme value that is a single character enclosed in single quotes. A character constant is any printable ASCII character (ASCII values between decimal value 32 and 126, or octal 40 and 176) but a character constant *cannot be* a single quote ('''), or backslash ('\') character. Character constants also include the character sequences, e.g. '"' to denote a quote, '\'' to denote a single quote, '\\' to denote backslash. All the escape sequences are listed in Section 4.4.

When scanning single quoted character constants **charConstant** or double quoted string constants: **stringConstant** you should ensure that:

- Character constants cannot contain more than one character, except in character constants that are escaped (see Section 4.4), e.g. 'an' is not valid, but '\n' is valid.
- String constants can have escape sequences (see Section 4.4).
- String and character constants are only terminated by an unescaped quote matching the open quote. In particular the following character constants should be treated as an error:
  - Character constants containing zero characters ''
  - Character constants with invalid closing delimiter, e.g. '\'
  - String constants with invalid closing delimiter, e.g. "\"

  The above rules indicate that the lexical analyzer should not simply truncate the token after a single character if no closing quote character is found.
- Unterminated string and character constants must be reported as errors.
- Invalid escape sequences or newlines embedded in string or character constants, should be reported as an error.

Some examples:

```
"\x" = STRING(error: unknown escape sequence)
"""  = STRING(""), STRING(error: unterminated string)
"
"    = STRING(error: newline in string constant)
'ab' = CHAR(error: char constant length greater than one)
'\'  = CHAR(error: unterminated char constant)
```

4.4 Escape Sequences

An escape sequence or escaped symbol means having a preceding backslash character to distinguish a special purpose character or to distinguish the character from a previously defined delimiter. The list of escape sequences are: *\t, \v, \r, \n, \a, \f, \b, *

Character constants also have *\'* defining an escaped open or close character delimiter. String constants also have *\"* defining an escaped open or close string delimiter. Invalid escape sequences should be reported as errors.

Some of the escape sequences have a special meaning: '`\t`' denotes a horizontal tab, '`\v`' denotes a vertical tab, '`\r`' denotes carriage return, '`\n`' denotes a newline, '`\a`' denotes an alert (bell) sent to the terminal, '`\f`' denotes a form feed (from back when terminal output was printed on paper), '`\b`' denotes a backspace.

You need to handle these escaped characters in the lexical definition, but you will not need to perform any special handling to implement their special roles. That will be done by the interaction of the standard input-output library and the terminal application.

5 Reference Grammar

The following reference grammar defines the structure of **`Decaf`** programs. It uses the notation defined in Section 3. This reference grammar is **not** a context-free grammar, although it can be easily converted into one.

⟨program⟩ → **class** ⟨class-name⟩ **'{'** ⟨field-decl⟩ * ⟨method-decl⟩ * **'}'**

⟨class-name⟩ → **id**

⟨field-decl⟩ → ⟨type⟩ { **id** | { **id** **'['** **intConstant** **']'** } } +, **';'**
| ⟨type⟩ **id** **'='** ⟨constant⟩ **';'**

⟨method-decl⟩ → { ⟨type⟩ | **void** } **id** **'('** [{ ⟨type⟩ **id** } +,] **')'** ⟨block⟩

⟨block⟩ → **'{'** ⟨var-decl⟩ * ⟨statement⟩ * **'}'**

⟨var-decl⟩ → ⟨type⟩ { **id** } +, **';'**

⟨type⟩ → **int** | **bool**

⟨statement⟩ → ⟨assign⟩ **';'**
| ⟨method-call⟩ **';'**
| **if** **'('** ⟨expr⟩ **')'** ⟨block⟩ [**else** ⟨block⟩]
| **while** **'('** ⟨expr⟩ **')'** ⟨block⟩
| **for** **'('** { ⟨assign⟩ } +, **';'** ⟨expr⟩ **';'** { ⟨assign⟩ } +, **')'** ⟨block⟩
| **return** [⟨expr⟩] **';'**
| **break** **';'**
| **continue** **';'**
| ⟨block⟩

⟨assign⟩ → ⟨lvalue⟩ **'='** ⟨expr⟩

⟨method-call⟩ → ⟨method-name⟩ **'('** [{ ⟨expr⟩ }$^{+}$**,**] **')'**
| **callout '('** **stringConstant** [{ **','** { ⟨callout-arg⟩ }$^{+}$**,** }] **')'**

⟨method-name⟩ → **id**

⟨callout-arg⟩ → ⟨expr⟩ | **stringConstant**

⟨lvalue⟩ → **id**
| **id '['** ⟨expr⟩ **']'**

⟨expr⟩ → ⟨lvalue⟩
| ⟨method-call⟩
| ⟨constant⟩
| ⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
| **'-'** ⟨expr⟩
| **'!'** ⟨expr⟩
| **'('** ⟨expr⟩ **')'**

⟨bin-op⟩ → ⟨arith-op⟩ | ⟨rel-op⟩ | ⟨eq-op⟩ | ⟨cond-op⟩

⟨arith-op⟩ → **'+'** | **'-'** | **'*'** | **'/'** | **'<<'** | **'>>'** | **'%'** | **rot**

⟨rel-op⟩ → **'<'** | **'>'** | **'<='** | **'>='**

⟨eq-op⟩ → **'=='** | **'!='**

⟨cond-op⟩ → **'&&'** | **'||'**

⟨constant⟩ → **intConstant** | **charConstant** | ⟨bool-constant⟩

⟨bool-constant⟩ → **true** | **false**

To help grasp some of the differences from other programming languages that you may be used to, here are some fragments of invalid code in **Decaf**. Check for yourself using the reference grammar above exactly *why* each of these examples are invalid.

```
class foo { int a; int b = a; } // Invalid!
```

```
int foo() { int a = 10; } // Invalid!
```

```
for(; a < b; ) // Invalid!
```

Of course, the reference grammar for **Decaf** could be changed to accept the examples above, *but do not change the grammar in any way* just because you feel it should accept these or other examples.

6 Semantics

A **Decaf** program consists of a single class declaration associated with an identifier. The class declaration consists of field declarations and method (or function) declarations. Field declarations introduce variables that can be accessed globally by all methods in the program.

6.1 Types

There are two basic types in **Decaf** – **int** for integers and **bool** for booleans. In addition, there are arrays of integers: **int id '['** *n* **']'** there are arrays of booleans: **bool id '['** *n* **']'** where *n* is an **intConstant** integer.

Arrays are declared only in the global (class declaration) scope. All arrays are one-dimensional and have a compile-time fixed size. Arrays are indexed from 0 to $n-1$, where $n > 0$ is the size of the array. The usual bracket notation is used to index arrays. Since arrays have a compile-time fixed size and cannot be declared as method parameters (or local variables), there is no facility to query the length of an array variable in **Decaf**.

6.2 Expressions

Expressions follow the normal rules from other languages like *C, C++* or *Java* for evaluation.

Integer constants evaluate to their integer value. Character constants evaluate to their integer ASCII values, e.g. **'A'** evaluates to the integer value 65 (consult `man ascii` for the full ASCII table). Note that **Decaf** does not have an explicit character type, instead we use the type **int** for characters.

An expression that refers to an array location, e.g. `x[10]` evaluates to the value contained at that location.

Method invocation expressions are discussed in Section 6.3.

Relational operators are used to compare integer expressions. The equality operators **'=='** and **'!='** are defined for **int** and **bool** types and can be used to compare any two expressions having the same type.

The result of a relational operator or equality operator has type **bool** .

The boolean connectives **'&&'** and **'||'** are interpreted using short circuit evaluation as in Java. This means: the side-effects of the second operand are not executed if the result of the first operand determines the value of the whole expression (i.e. if the result is **false** for **'&&'** or **true** for **'||'**).

Precedence	Operators	Explanation
1	'-'	unary minus
2	'!'	logical not
3	'*' '/'	multiplication, division
4	'+' '-'	addition, subtraction
5	'%'	modulus op
6	'<<' '>>' 'rot'	bit shift ops
7	'<' '<=' '>=' '>'	relational ops
8	'==' '!='	equality
9	'&&'	conditional and
10	'||'	conditional or

The precedence level for each operator is shown in the table above. All operators at the same precedence level get equal precedence. All operators with equal precedence associate left.

Binary **'%'** computes the modulus of two numbers. Given integer operands `a` and `b`: If `b` is positive, then `a % b` is `a` minus the largest multiple of `b` that is not greater than `a`. If `b` is negative, then `a % b` is `a` minus the smallest multiple of `b` that is not less than `a` (i.e. the result will be less than or equal to zero).

Binary **'rot'** rotates the bits of the first argument by the number of bits specified by the second argument. If the second argument is a positive number then rotate left, and if it is negative then rotate right by the absolute amount of the second argument.

Number constants in **Decaf** are either decimals or hexadecimals. Decimal numbers in **Decaf** are 32-bit signed integers between the values -2147483647 to 2147483647. However, range checking for 8-digit hex constants is

based on unsigned 32-bit integers, even though hex values greater than 2147483647_{10} are actually negative (hex `0xffffffff` is -1). The reason for not bothering with the sign for hex digits is that they are used as bit patterns without regard for numeric value.

6.3 Method Calls and Callouts

The program must contain a declaration for a method called **main** that has no parameters. The return type of the method **main** has to be the type **void** . Execution of a **Decaf** program starts at this method **main** . Other methods defined as part of the class declaration can have zero or more parameters and must have a return type explicitly defined.

Callout functions using the **callout** keyword are used to invoke external library functions. The most useful library functions that you will use are the `print_int` and `read_int` functions. The `print_int` callout function is invoked as: `callout("print_int", z);` where `z` is an integer variable that has a value. The `read_int` callout function is invoked as: `z = callout("read_int");` where integer variable `z` receives the result of calling the `read_int` library function.

7 Brief History of Decaf

Decaf has been used as part of Compilers courses in several universities including Stanford University, MIT, University of Delaware, Southern Adventist University, University of Tennessee, among others. The precise genesis of **Decaf** is not entirely clear. Some believe it was a revision of the SOOP language developed by Maggie Johnson and Steve Freund at Stanford. Others believe it was a simplification of a language called **Espresso** used at MIT. Still others claim that **Decaf** was invented at the University of Tennessee. In any case, **Decaf** is a useful language for introductory compiler courses. Our version of **Decaf** as described in this document is distinct and quite different, apart from its general structure, from all the other versions of Decaf.