CMPT 379: **Decaf** Language Definition

Anoop Sarkar – anoop@cs.sfu.ca

January 25, 2004

The programming assignments for CMPT 379 will build various components towards a working compiler for a programming language called **Decaf**. This document is the specification for the **Decaf** language. This language definition will 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.

1 Sample Decaf Code

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

```
class GreatestCommonDivisor {

    void main() {
       int x, y, z;
       x = 10;
       y = 20;
       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) );
       }
    }

}
```

2 Notation

⟨foo⟩	means ⟨foo⟩ is a non-terminal
foo	means **foo** is a terminal, 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 **‘;’**
$[x]$	means zero or one occurence of x, i.e. x is optional n.b.: do not confuse this notation with terminals **‘[’** and **‘]’**
x^*	zero or more occurences of x
x^+	one or more occurences of x
$\{\}$	curly braces are used for grouping n.b.: do not confuse this notation with terminals ‘{’ and ‘}’

3 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.

3.1 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”.

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).

Identifiers, 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 when defining the grammar of **Decaf**.

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

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.

String constants, denoted by the token **stringConstant** will have a lexeme value that is composed of characters enclose in double quotes. Strings can contain any character except a newline or a double quote. A string must start and end on a single line, it cannot be split over multiple lines.

Character constants, denoted by the token **charConstant** will have a lexeme value that is a single character enclosed in single quotes.

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`

A character constant, denoted by **charConstant**, is any printable ASCII character (ASCII values between decimal value 32 and 126, or octal 40 and 176) other than quote (’"’), single quote (’’’), or backslash (’\’), plus the two character sequences ’\"’ to denote a quote, ’\’’ to denote a single quote, ’\\’ to denote backslash, ’\t’ to denote a tab constant or ’\n’ to denote a newline.

3.2 Token Boundaries

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.

Here are some examples that explain these rules:

```
0x123food  = HEX(0x123f), IDENT(ood)
0xfood123  = HEX(0xf), IDENT(ood123)
123break   = INT(123), KW_BREAK
0x123rot3  = HEX(0x123), IDENT(rot3)
0x123rot 3 = HEX(0x123), KW_ROT, INT(3)
break123   = IDENT(break123)
breakwhile = IDENT(breakwhile)
```

3.3 String and Character Constants

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

- String and character constants are only terminated by an unescaped quote matching the open quote. In particular, character constants containing either zero or more than one character that are not escaped (i.e. having a backslash character) should be treated as a single token, with an error. Do not simply truncate the token after a single character is no closing quote character is found.
- Unterminated string and character constants must be specifically reported as errors, with the starting line number
- Invalid escape sequences (i.e. having a backslash character) or newlines embedded in string or character constants, should be reported once the token boundaries have been determined.

- Similarly, unescaped single quotes found inside a string constant or unescaped double quotes found inside a character constant, must be reported once the token boundaries have been determined.

Some examples:

```
"'"  = STRING(error: unescaped singlequote)
"\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 Semantics

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.

5 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. While the origins of **Decaf** are shrouded in mystery, it continues to be a useful language for introductory compiler courses in universities around the world.