Syntax of the Brown Decaf Programming Language*

Steve Reiss
CS 126 Spring 2006

1 Introduction

This document describes the syntax of the programming language Decaf. Decaf is a subset of Java containing the essential features of classes and objects but without many of the more complex features such as threads and exception handling.

The final project for CS 126 is to write a compiler that compiles Decaf programs into x86 assembly code. This document contains all the information you need to write a lexical analyzer and parser for Decaf. The information necessary to complete the project will follow in a second document on the semantics of Decaf.

Please report any bugs, ambiguities, or other problems with this document to `spr@cs.brown.edu`.

2 Lexical Components

The first step of compiling a Decaf program is to convert the input stream of characters into an input stream of tokens, where each token corresponds to a Decaf lexeme. The lexemes of a Decaf program are keywords, primitive type names, literals, punctuation, comments, and identifiers. Each of these is discussed in the following sections.

*Based on the document *Syntax of the Decaf Programming Language*, by Dan DuVarney and Purush Iyer at the North Carolina State University with help from Manos Renieris at Brown.

2.1 Notation

The following notation is used in describing the lexemes:

ϵ	represents the empty string
X^*	represents zero or more occurrences of X
X^+	represents one or more occurrences of X
$X^?$	represents zero or one occurrences of X

2.2 Whitespace

Whitespace makes programs easier to read by humans and is not tokenized. The whitespace characters are space, tab, and newline.

2.3 Keywords and Forbidden Words

Decaf has the following keywords:

```
break   class   continue  else       extends
if      new     private   protected  public
return  static  super     this       while
```

The case of keywords is significant. `if` is a keyword. `If` is not. Additionally, any Java keyword which is not a Decaf keyword is a forbidden word. Forbidden words cannot be used in a Decaf program. The purpose of this rule is to make it easy to convert Decaf programs into legal Java programs. The forbidden Decaf words are:

```
abstract  byte          case    catch       const
default   do            double  final       finally
for       implements    import  instanceof  interface
long      native        goto    package     short
switch    synchronized  throw   throws      transient
try       volatile
```

The following words are also reserved for possible future extension to Java and are forbidden in Decaf:

```
byvalue  cast      future  generic  inner
none     operator  outer   rest     var
```

Your lexical analyzer should generate an error message if any forbidden word appears in a Decaf program, and the program should be rejected.

2.4 Identifiers

A Decaf identifier is a letter or underscore character (`_`) followed by zero or more letters, digits, and underscore characters, which is not a keyword or forbidden word. Note that when matching lexemes, the longest match should always be chosen. For example, `ifelse` is an identifier, not the keyword `if` followed by the keyword `else`. Identifiers are also case sensitive, so `Aaa` and `AAa` are different identifiers.

2.5 Comments

Comments are ignored by the lexical analyzer. Decaf has two styles of comments:

`/* comment */`	All characters from `/*` until the first occurrence of `*/` are ignored (just like C and C++).
`// comment`	All characters from the `//` until the end of line are ignored.

Note that `//` is ignored when it appears inside of `/*` and `*/`. Hence, the following is a well-terminated comment:

```
/* this is a comment
   there is a // but it's ignored */
```

2.6 Primitive Types

The names of the Decaf primitive types are recognized by the lexical analyzer in a manner similar to keywords. These names are:

```
boolean  char  int  void
```

Names of Java primitive types which aren't supported in Decaf are treated as forbidden words. The unsupported types are:

```
byte  double  float  long  short
```

2.7 Literals

There are five kinds of literals allowed in Decaf.

2.7.1 Integer Literals

An integer literal is `0` or a non-zero base-10 digit followed by zero or more base-10 digits. The value of an integer literal is the standard base-10 interpretation. Some sample integer literals are:

Literal	Value
`1273`	1273_{10}
`9`	9_{10}
`10000`	10000_{10}
`0`	0

2.7.2 Floating-Point Literals

Support of floating-point literals is not required.

2.7.3 Character Literals

A character literal is any of the following:

- `'`x`'` where x is any character other than backslash (`\`), ASCII newline, or single quote (`'`). The literal value is the character x.
- `'\`x`'` where x is any character other than `n` or `t`. The literal value is the character x.
- `'\t'` — The literal value is the ASCII tab character.
- `'\n'` — The literal value is the ASCII newline character

Some examples of character literals are:

Literal	Value
`' '`	a single space
`'\n'`	a newline (ASCII LF) character
`'x'`	the character `x`
`'\\'`	the character `\`

2.7.4 String Literals

A string literal is a double quote (") followed by a sequence of characters and ended with another double quote ("). The characters that may appear within a string literal are restricted as follows:

1. The newline character cannot appear.
2. The double quote (") character cannot appear, except when preceded by an odd number of backslash characters (\).
3. The string is ended by the first double quote not preceded by an odd number of backslash characters.
4. The semantics of backslash are the same as in character literals.

Some sample string literals are:

Literal	**Value**
`"\"hello\\"`	`"hello\`
`"abcde"`	`abcde`
`"this is a test"`	`this is a test`

2.7.5 Boolean Literals

The boolean literals are `true` and `false`.

2.7.6 Null Literals

The only null literal is the word `null`.

2.8 Punctuation

The following characters are Decaf punctuation:

```
( ) { } [ ] ; , .
```

2.9 Operators

The following character sequences are Decaf operators:

```
=   >   <   !
==  >=  <=  !=
+   -   *   /
&&  ||  %
```

The following character sequences are Java operators that aren't supported in Decaf:

```
~   ?   :    ++   --
&   |   ^    <<   >>    >>>
+=  -=  *=   /=   &=    |=
^=  %=  <<=  >>=  >>>=
```

These operators are forbidden and should trigger a compile error.

2.10 Other Characters

Any input not conforming to the rules in this section is illegal and should generate an error.

3 Decaf Grammar

All Decaf programs must conform to the following grammar.

3.1 Notation

The terminal symbols used in this description of the Decaf grammar are:

Category	**Symbols**
Identifiers	*identifier*
Literals	*intLiteral charLiteral booleanLiteral*
Keywords	`if while else` ...
	(See section 2.3 for a complete list).
Primitive Types	`boolean char int void`
Punctuation	`( ) { } [ ] ; , .`
Operators	`+ - * / =` ...
	(See section 2.9 for a complete list).

In addition to these nonterminals, the following notation is used:

Class	is a non-terminal symbol (the first letter is capitalized)
ϵ	represents the empty string
X^*	represents zero or more occurrences of X
X^+	represents one or more occurrences of X
$X^?$	represents zero or one occurrences of X
$X \rightarrow Y$	represents a production
$X \rightarrow Y \mid Z$	is shorthand for $X \rightarrow Y$ or $X \rightarrow Z$

3.2 Decaf Grammar Productions

The productions in the Decaf Grammar are:

Start → *Class*$^+$

Class → `class` *identifier Super*$^?$ `{` *Member** `}`

Super → `extends` *identifier*

Member → *Field* | *Method* | *Ctor*

Field → *Modifier** *Type VarDeclaratorList* `;`

Method → *Modifier** *Type identifier FormalArgs Block*

Ctor → *Modifier** *identifier FormalArgs Block*

Modifier → `static` | `public` | `private` | `protected`

FormalArgs → `(` *FormalArgList*$^?$ `)`

FormalArgList → *FormalArg*
FormalArgList → *FormalArg* `,` *FormalArgList*

FormalArg → *Type VarDeclaratorId*

Type → *PrimitiveType*
Type → *identifier*

Type → *Type* `[]`

PrimitiveType → `boolean` | `char` | `int` | `void`

VarDeclaratorList → *VarDeclarator* `,` *VarDeclaratorList*
VarDeclaratorList → *VarDeclarator*

VarDeclarator → *VarDeclaratorId*
VarDeclarator → *VarDeclaratorId* `=` *Expression*

VarDeclaratorId → *identifier*
VarDeclaratorId → *VarDeclaratorId* `[]`

Block → `{` *Statement** `}`

Statement → `;`
Statement → *Type VarDeclaratorList* `;`
Statement → `if (` *Expression* `)` *Statement*
Statement → `if (` *Expression* `)` *Statement* `else` *Statement*
Statement → *Expression* `;`
Statement → `while (` *Expression* `)` *Statement*
Statement → `return` *Expression*$^?$ `;`
Statement → `continue ;`
Statement → `break ;`
Statement → `super` *ActualArgs* `;`
Statement → *Block*

Expression → *Expression BinaryOp Expression*
Expression → *UnaryOp Expression*
Expression → *Primary*

BinaryOp → `=` | `||` | `&&` | `==` | `!=` | `<` | `>` | `<=` | `>=` | `+` | `-` | `*` | `/` | `%`

UnaryOp → `+` | `-` | `!`

Primary → *NewArrayExpr*
Primary → *NonNewArrayExpr*
Primary → *identifier*

NewArrayExpr → `new` *identifier Dimension*$^+$
NewArrayExpr → `new` *PrimitiveType Dimension*$^+$

Dimension → `[` *Expression* `]`

NonNewArrayExpr → *Literal*
NonNewArrayExpr → `this`
NonNewArrayExpr → `(` *Expression* `)`
NonNewArrayExpr → `new` *identifier ActualArgs*
NonNewArrayExpr → *identifier ActualArgs*
NonNewArrayExpr → *Primary* `.` *identifier ActualArgs*
NonNewArrayExpr → `super` `.` *identifier ActualArgs*
NonNewArrayExpr → *ArrayExpr*
NonNewArrayExpr → *FieldExpr*

FieldExpr → *Primary* `.` *identifier*
FieldExpr → `super` `.` *identifier*
ArrayExpr → *identifier Dimension*
ArrayExpr → *NonNewArrayExpr Dimension*

Literal → `null` | `true` | `false` | *intLiteral* | *charLiteral* | *stringLiteral*

ActualArgs → `(` *ExprList*$^?$ `)`

ExprList → *Expression*
ExprList → *Expression* `,` *ExprList*

3.2.1 Dangling Else

In Decaf, the `else` keyword always binds to the most recent `if`. Hence, `if` C_1 `if` C_2 S_1 `else` S_2 is equivalent to `if` C_1 `{ if` C_2 S_2 `else` S_2 `}`.

3.2.2 Operator Precedence

The Decaf operators have the following precedence rules:

1. Unary operators have precedence over binary operators.

2. The precedence of binary operators is given by the following table (1 is the highest precedence, 7 lowest):

Operator	**Precedence**	**Associativity**
`!`	0	None
`Unary -`	0	None
`Unary +`	0	None
`*`	1	Left
`/`	1	Left
`%`	1	Left
`+`	2	Left
`-`	2	Left
`<`	3	Not Associative
`>`	3	Not Associative
`<=`	3	Not Associative
`>=`	3	Not Associative
`==`	4	Left
`!=`	4	Left
`&&`	5	Left
`||`	6	Left
`=`	7	Right

For example, the expression

```
a * b + c = a - 2 == f = 4
```

should be parsed as

```
((a * b) + c) = (((a - 2) == f) = 4)
```