Definición del Lenguaje **Decaf**

Iván de Jesús Deras Tábora – ideras@gmail.com

25 Enero del 2019

1 Introducción

El proyecto de la clase consistirá en construir un compilador para un lenguaje llamado *Decaf*. *Decaf* es un lenguaje fuertemente tipificado, orientado a objetos con soporte para herencia y encapsulamiento. El diseño de *Decaf* tiene ciertas similitudes con otros lenguajes como C, C++ o Java.

Aunque debemos tener en cuenta que *Decaf* no es una copia exacta de ninguno de esos lenguajes. La funcionalidad de *Decaf* ha sido reducida considerablemente comparado con un lenguaje "completo". Esto se hizo para hacer el proyecto de la clase realizable en un semestre (dos trimestres). A pesar de esto *Decaf* será capaz de ejecutar programas complicados.

2 Programa Ejemplo **Decaf**

El siguiente es un programa escrito en *Decaf*:

```
class GreatestCommonDivisor {
      int a = 10;
      int b = 20;
      void main() {
            int x, y, z;
            x = a;
            y = b;
            z = gcd(x, y);

            System.out.println(z);
      }

      // Función que calcula el máximo común divisor
      int gcd(int a, int b) {
            if (b == 0) { return(a); }
            else { return( gcd(b, a % b) ); }
      }
}
```

3 Notación

<nts>	Significa que <nts> es un símbolo no terminal.
ts	Significa que **ts** es un símbolo terminal, o sea un Token reconocido por el analizador léxico.
'x'	Significa que la cadena *x* es un terminal cuyo lexema es *x*.
[*x*]	Significa cero o más ocurrencias de *x*.
*x**	Significa cero o más ocurrencias de *x*.
x+	Significa una o más ocurrencias de *x*.
{*x*}+,	Una lista separada por comas de una o más ocurrencias de x
{ }	Las llaves son usadas para agrupar
|	Usado para separar alternativas
[c1-c2]	Denota un conjunto de caracteres. Ej: [a-c] denota los tres caracteres a, *b, c*

4 Consideraciones Léxicas

Todas las palabras reservadas son minúsculas. Las palabras reservadas y los identificadores son case-sensitive. Por ejemplo, *if* es una palabra reservada, pero *IF* es un identificador. También, compi1 y Compi1 son dos identificadores diferentes.
Los comentarios linea comienzan con **//** y terminan con un fin de línea y los comentarios de bloque comienzan con **/*** y terminan con ***/.**

4.1 Definición de Tokens

Las palabras reservadas son las siguientes:

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

Por ahora las palabras reservadas **extends, new** y **null** no aparecerán en ninguna parte de la definición del lenguaje; estas palabras están reservadas para uso futuro.

Los operadores y tokens de puntuación son los siguientes:

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

Tokens como **stringConstant** el cual define una constante cadena como `"hello, world"` o tokens de un solo carácter, tales como **';'** o **'.'** no aparecen en la lista anterior pero son tokens válidos y son usados en la definición de la gramática de *Decaf* (Vea la sección 5).

Los identificadores, denotados por el token **ID** son definidos como una cadena de caracteres que comienza con un carácter alfabético o un guión bajo ([a-zA-Z_]) seguido de cero o más caracteres alfanuméricos incluyendo el guión bajo ([a-zA-Z_0-9]).

Las palabras reservadas y los identificadores deben estar separados por espacios en blanco, o un token que no sea una palabra reservada o un identificador.

Ej: ***thiswhiletrue*** es un solo identificador, no tres distintas palabras reservadas. Veas la sección 4.2 para algunos ejemplos.

Constantes de cadena, denotadas por el token **stringConstant** tendrán un valor léxico compuesto de caracteres encerrados entre comillas dobles. Una cadena debe comenzar y terminar en una sola línea, no puede expandirse en multiples lineas. Hay que tener en cuenta que una constante de cadena puede incuir caracteres de escape como \n. Para más detalles sobre cadenas y caracteres vea la sección 4.3

Contantes Entero en Decaf, denotadas por el token **intConstant**, son decimales (base 10) o hexadecimales (base 16). Una constantes entera hexadecimal comienza con **0x** seguido de una secuencia de dígitos hexadecimales ([0-9A-Fa-f]. Ejemplos de constantes enteras: 8, 012, 0x0, 0x12aE

Constantes Carácter, denotados por el token **charConstant** tendrán un valor léxico que es un solo character encerrado entre comillas sencillas. Una constante carácter es cualquier caraćter ASCII que sea imprimible (Valores ASCII enter 32y 126). Una constante carácter no puede ser una comilla sencilla **'''** ni tampoco una barra invertida '\'. Para más detalles sobre cadenas y caracteres vea la sección 4.3. Las constantes de carácter tambien incluyen secuencias de caracteres, por ejemplo '\'' denota el carácter de comilla sencilla, '\\' representa el carácter de barra invertida. Todas las secuencias de escape se listan en la sección 4.4.

4.2 Reconocimiento de Tokens y Espacios en Blanco

El reconocimiento de tokens tales como constants enteras, palabras reservadas e identificadores son explicadas usando las siguientes reglas. En efecto estas reglas definen un algoritmo para agrupar caracteres del conjunto [a-zA-Z0-9] en tokens.

- Si la secuencia comienza con **0x**, entonces estos dos caracteres y la secuencia más larga de caracteres del conjunto [0-9a-fA-F] que sigue forman un dígito hexadecimal. El último carácter de esa secuencia marca el fin del token.

- Si la secuencia comienza con un dígito decimal (pero no **0x**) entonces la secuencia más larga de dígitos decimales forman una constante entera.
  Tenga en cuenta que la semántica de verificación de rango se llevará a cabo luego, de esta forma la secuencia 123456789123456789 la cual claramente está fuera de rango, será reconocida como un solo token por el lexer.

- Si la secuencia comienza con un carácter alfabetico, entonces este carácter y la secuencia más larga de caracteres alfanuméricos [0-9a-zA-Z_] que siguen a éste carácter inicial forman un token que puede ser un identificador o una palabra reservada.

- Espacios en blanco y otras definiciones de tokens juegan un papel importante en la delimitación de tokens. Por ejemplo la cadena **while3** es un solo identificador, pero **while 3** son dos tokens, la palabra reservada **while** y el token constante entera **3,** **while(3)** representa 4 tokens, la palabra reservada **while**, el paréntesis izquierdo, la

constante entera **3**, y el paréntesis derecho. Se consideran espacios en blanco los caracteres \n, \t, \r y el carácter ASCII de espacio.

Aquí hay varios ejemplos de las reglas explicadas anteriormente:

0x123food = CONST_ENTERA(0x123f), IDENTIFICADOR(ood)
0xfood123 = CONST_ENTERA(0xf), IDENTIFICADOR(ood123)
123break = CONST_ENTERA(123), KW_BREAK
0x123while3 = CONST_ENTERA(0x123), IDENTIFICADOR(while3)
0x123while 3 = CONST_ENTERA(0x123), KW_WHILE, CONST_ENTERA(3)
1250x356 = CONST_ENTERA(1250), IDENTIFICADOR(x356)
break123 = IDENTIFICADOR(break123)
breakwhile = IDENTIFICADOR(breakwhile)

4.3 Constantes de cadena y carácter

Las constantes de cadenas, denotadas por el token **stringConstant** tendrán un valor léxico compuesto de los caracteres encerrados entre comillas dobles. Una cadena debe comenzar y terminar en una sola línea, no puede dividirse en múltiples líneas. Hay que destacar que una constante de cadena puede contener secuencias de escape como \n.

Las constantes de carácter, denotadas por el token **charConstant** tendrán un valor léxico que es un solo carácter encerrado entre comillas sencillas. Una constante carácter es cualquier carácter ASCII que tenga representación gráfica (Valores entre 32 y 126). Una constante carácter no puede ser una comilla sencilla **‘’’** o una barra invertida ‘\’. Las constantes de carácter pueden incluir secuencias de escape, por ejemplo ‘\’’ para representar una comilla sencilla, ‘\\’ para representar una barra invertida. Todas las secuencias de escape se describen en la sección 4.4.
Al momento de reconocer constantes de carácter **charConstant** o constantes de cadena **stringConstant** en el analizador léxico, deberá asegurarse que:

- Las constantes de carácter no pueden contener más de un carácter, excepto en constantes de carácter que usan secuencias de escape (ver sección 4.4), ejemplo ‘an’ no es válido, pero ‘\n’ si es válido.
- Las constantes de cadena pueden contener secuencias de escape (ver sección 4.4)
- Las constante de cadena y de carácter deberán cerrarse por una comilla doble o sencilla, en particular las siguientes constantes deberán tratarse como un error:
    - Constantes de carácter que contengan cero caracteres ‘’
    - Constantes de carácter con carácter de cierre inválido, ‘\’
    - Constantes de cadena con carácter de cierre inválido, “abc\”
- Constantes de cadena y de carácter sin el carácter de cierre deben ser reportadas como errores.
- Secuencias de escape inválidas deben reportarse como errores, por ejemplo:
    - "\x" = STRING(error: secuencia de escape desconocida)
    - “””= STRING(""), STRING(error: cadena no terminada)
    - ‘ab’ = CHAR(error: Constante carácter de longitud mayor a uno)
    - ‘\’ = CHAR(error: Constante carácter no terminada)

4.4 Secuencias de Escape

Una secuencia de escape significa tener una barra invertida precediendo a un carácter con un propósito especial. Las lista de secuencias de escape son: \t, \r, \n, \\. Las constantes de carácter pueden tener \' como secuencia de escape para denotar la comilla sencilla. Las constantes de cadena pueden tener \" como secuencia de escape para denotar la comilla doble. Las secuencias de escape inválidas deben reportarse como errores.

Algunas de las secuencias de escape tienen un significado especial: \t denota un tab horizontal, \r denota un retorno de carro, \n denota una nueva linea.

Usted deberá manejar todas estas secuencias de escape en la definición léxica, pero no ejecutará ningún tratamiento especial para implementar su significado. Esa implementación se hará cuando se implemente el interprete del lenguaje.

5 Gramática de Referencia

La siguiente grmática de referencia define la estructura de un programa de Decaf. Aquí usamos la notación definida en la sección 3.
Esta gramática de referencia no es libre de contexto, aunque podría convertirse fácilmente en gramática libre de contexto.

```
<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> -> id ‘(‘ [{ <expr> }+,] ‘)’
              | System.out.print ‘(‘ <argument> ‘)’
              | System.out.println ‘(‘ <argument> ‘)’
              | System.in.read ‘(‘ ‘)’
              | Random.nextInt ‘(‘ <expr> ‘)’

<argument> -> stringConstant
              | <expr>

<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> -> ‘+’ | ‘-‘ | ‘*’ | ‘/’ | ‘<<’ | ‘>>’ | ‘%’

<rel-op> -> ‘<’ | ‘>‘ | ‘<=’ | ‘>=’

<eq-op> -> ‘==’ | ‘!=‘

<cond-op> -> ‘&&’ | ‘||‘

<constant> -> intConstant | charConstant | <bool-constant>

<bool-constant> -> true | false
```

Para ayudarles a entender algunas de las diferencias de otros lenguajes de programación que usted haya usado (como ser Java, C++, o C#), aquí están algunos fragmentos de código que no son válidos en *Decaf*. Usted puede verficarlo usando la gramática descrita anteriormente.

```
class foo { int a; int b = a; } // No valido!
int foo() { int a = 10; } // No valido!
for(; a < b; ) // No valido!
```

Está claro que la gramática de *Decaf* podría cambiarse para aceptar dichos ejemplos, pero NO DEBE cambiar la gramática de ninguna manera.

6 Reglas Semanticas

Un programa de *Decaf* consiste de una definición de una clase asociada con un identificador. La declaración de la clase consiste de declaraciones de campos y declaraciones y métodos (funciones).
Las declaraciones de campos definen variables que pueden ser accesadas globalmente por todos los métodos en el programa.

6.1 Tipos de Datos

Hay tres tipos básicos en *Decaf* – **int** para enteros con signo de 32 bits, **bool** para valores booleanos y **char** para caracteres de 1 byte. Además hay arreglos de enteros y booleanos. Los arreglos son declarados solamente en el escope global. Todos los arreglos son de una dimensión y tienen un tamaño fijo definido en tiempo de compilación. Los arreglos son indexados comenzando en cero hasta n-1, donde n > 0 es el tamaño del arreglo. Utilizamos la notación de brackets para accesar los elementos del arreglo.

6.2 Expresiones

Las expresiones siguen las reglas normales de otros lenguajes como C, C++ o Java para el orden de evaluación. Las constantes enteras evaluan a su valor entero. Las contantes carácter evaluan a su valor entero ASCII, ej. 'A' evalua a 65 (si tiene Linux puede ejecutar *man ascii* para la tabla ASCII completa)
Una expresión que hace referencia a un elemento de un arreglo, ej. x[10] evalua al valor contenido en la posición referenciada.
Expresiones que involucran llamadas a métodos son discutidas en la sección 6.3.
Los operadores relacionales son usados para comparer expresiones con enteros. Los operadores de igualdad '==' y '!=' están definidos para los tipos **int** y **bool** y pueden ser usados para comparar dos expresiones que tengan el mismo tipo.
El resultado de un operador relacional o de igualdad tiene tipo **bool**.
Los operadores condicionales **'&&'** y **'||'** son interpretados usando corto-circuito asi como en Java. Esto significa que los efectos secundarios del segundo operando no son ejecutados si el resultado del primer operando determina el valor de la expresión (Esto es: si el resultado es **falso** para '&&' o **true** para **'||'**).

Explicación de la precedencia de los operadores

1	'-'	Menos unario
2	'!'	Negación lógica
3	'*' '/'	Multiplicación, división
4	'+' '-'	Suma, resta
5	'%'	Operación Módulo
6	'<<' '>>'	Operadores de corrimiento de bits
7	'<' '<=' '>=' '>'	Operadores relacionales
8	'==' '!='	Igualdad
9	'&&'	Y condicional
10	'||'	O condicional

El nível de precedencia de cada operador se muestra en la tabla anterior. Todos los operadores en el mismo nivel tienen la misma precedencia. Los operadores con igual precedencia se asocian por la izquierda.
El operador binario '%' se utiliza para cálcular el módulo de dos números.

Las contantes numéricas en Decaf pueden ser decimals o hexadecimals. Los números decimales en *Decaf* son enteros de 32-bit con signo, cuyos valores van desde -2147483647 to 2147483647. Sin embargo, la verificación de rango para las constantes hexadecimal de 8-dígitos es basado en enteros sin signo de 32-bits, eso significa que valores mayores a 2147483647_{10} son valores

negativos (El valor hexadecimal 0xFFFFFFFF es -1). La razón por la cual ignoramos el signo en las constantes hexadecimal es porque usualmente se utilizan como patrones de bits, sin importar su valor.

6.3 Llamadas a Métodos

Todo programa deberá contener una declaración de un método llamado ***main*** el cual no deberá contener parámetros. El tipo de retorno del método ***main*** tiene que ser **void.** La ejecución de un programa de *Decaf* comienza en este método ***main***.
Otros métodos definidos como parte de la declaración de la clase pueden tener cero o más parámetros y deben tener un tipo de retorno definido de forma explícitia.

7 Breve Historia de *Decaf*

Decaf ha sido usado como parte de la clase de Compiladores en varias universidades incluyendo Standford, MIT, Universidad de Delaware, Universidad Adventista del Sur, Universidad de Tenessee, entre otras. El origen exacto de *Decaf* no está realmente claro. Algunos creen que es una revisión del lenguaje SOOP desarrollado por Maggie Johnson y Steve Freund en Standford. Otros dicen que es una siplificación del lenguaje Espresso usado en MIT. Y algunos dicen que *Decaf* fue inventado en la Universidad de Tennessee. De cualquier forma *Decaf* es útil como lenguaje para un curso de compiladores. Nuestra versión de *Decaf* descrita en este documento es distinta de otras versiones de *Decaf*, aunque conserva la estructura general.