Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada (DIMAp)

DIM0611 - COMPILADORES

Prof.: Martin Alejandro Musicante

Decaf - Documentação

Componentes:
Candinho Luiz Dalla Brida Junior (20180152552)
Débora Emili Costa Oliveira (20180008192)
Vinícius Campos Tinoco Ribeiro (20180153460)

13 de agosto de 2018

Sumário

1 Definição da Linguagem

A linguagem **Decaf** é implementada seguindo as orientações definidas pela Texas A&M University (TAMU) que pode encontrada neste site[1]. Desenvolvido como projeto da disciplina de Compiladores do curso de Ciência da Computação da UFRN.

A linguagem **Decaf** é fortemente tipada, orientada a objetos, com suporte a herança e encapsulamento. O design possui muitas semelhanças com C/C++/Java, porém com o conjunto de recursos reduzido e simplificado para manter os projetos de programação gerenciáveis. O projeto do compilador em desenvolvimento para programas escritos em **Decaf** está disponível no GitHub[2].

Um programa **Decaf** é uma sequência de declarações, onde cada declaração estabelece uma variável, função, classe ou interface. Um programa nesta linguagem deve ter uma função global chamada *main* que não recebe argumentos e não retorna valor algum. Esta função serve como ponto de entrada para a execução do programa.

1.1 ”Hello, World!”em Decaf e linguagens na qual foi baseado

Os Algoritmos a seguir mostram o código básico para imprimir o texto ”Hello, World!”. O objetivo é demonstrar a estrutura básica de um algoritmo escrito em Decaf, e evidenciar as diferenças e igualdades básicas entre Decaf e as linguagens da qual possui semelhanças.

Hello World em **Decaf**

```
void main() {
    print("Hello, World!");
}
```

Hello World em **Java**

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Hello World em **C**

```
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello World!\n");
    return (0);
}
```

Hello World em **C++**

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, World!";
    return 0;
}
```

[1] https://parasol.tamu.edu/courses/decaf/students/decafOverview.pdf
[2] https://github.com/Vinihcampos/decaf

2 Gramática

A sintaxe formal da linguagem em Backus-Naur Form (BNF) foi definida no projeto da TAMU e é apresentada a seguir:

⟨Program⟩ ⊨ ⟨Decl⟩$^+$
⟨Decl⟩ ⊨ ⟨VariableDecl⟩ | ⟨FunctionDecl⟩ | ⟨ClassDecl⟩ | ⟨InterfaceDecl⟩
⟨VariableDecl⟩ ⊨ ⟨Variable⟩;
⟨Variable⟩ ⊨ ⟨Type⟩ **ident**
⟨Type⟩ ⊨ **int** | **double** | **bool** | **string** | **ident** | ⟨Type⟩**[ ]**
⟨FunctionDecl⟩ ⊨ ⟨Type⟩ **ident (**⟨Formals⟩ **)**⟨StmtBlock⟩ | **void ident (** ⟨Formals⟩ **)** ⟨StmtBlock⟩
⟨Formals⟩ ⊨ ⟨Variable⟩$^+$, | **λ**
⟨ClassDecl⟩ ⊨ **class ident**⟨Extends⟩⟨Implements⟩ **{** ⟨Field⟩* **}**
⟨Extends⟩ ⊨ **extends ident** | **λ**
⟨Implements⟩ ⊨ **implements ident**$^+$, | **λ**
⟨Field⟩ ⊨ ⟨VariableDecl⟩ | ⟨FunctionDecl⟩
⟨InterfaceDecl⟩ ⊨ **interface ident {** ⟨Prototype⟩***}**
⟨Prototype⟩ ⊨ ⟨Type⟩ **ident (** ⟨Formals⟩ **) ;** | **void ident (** ⟨Formals⟩ **) ;**
⟨StmtBlock⟩ ⊨ **{** ⟨VariableDecl⟩*⟨Stmt⟩* **}**
⟨Stmt⟩ ⊨ ⟨Expr1⟩**;** | ⟨IfStmt⟩ | ⟨WhileStmt⟩ | ⟨ForStmt⟩ | ⟨BreakStmt⟩ | ⟨ReturnStmt⟩ | ⟨PrintStmt⟩ | ⟨StmtBlock⟩
⟨IfStmt⟩ ⊨ **if(** ⟨Expr⟩ **)** ⟨Stmt⟩⟨ElseStmt⟩
⟨ElseStmt⟩ ⊨ **else** ⟨Stmt⟩ | **λ**
⟨WhileStmt⟩ ⊨ **while(** ⟨Expr⟩ **)** ⟨Stmt⟩
⟨ForStmt⟩ ⊨ **for(** ⟨Expr1⟩;⟨Expr⟩;⟨Expr1⟩**;)** ⟨Stmt⟩
⟨Expr1⟩ ⊨ ⟨Expr⟩ | **λ**
⟨ReturnStmt⟩ ⊨ **return** ⟨Expr1⟩
⟨BreakStmt⟩ ⊨ **break;**
⟨PrintStmt⟩ ⊨ **print (**⟨Expr⟩$^+$, **);**
⟨Expr⟩ ⊨ ⟨Lvalue⟩=⟨Expr⟩ | ⟨Constant⟩ | ⟨LValue⟩ | **this** | ⟨Call⟩ | **(** ⟨Expr⟩ **)** | ⟨Expr⟩+⟨Expr⟩ | ⟨Expr⟩-⟨Expr⟩ | ⟨Expr⟩*⟨Expr⟩ | ⟨Expr⟩/⟨Expr⟩ | ⟨Expr⟩%⟨Expr⟩ | -⟨Expr⟩ | ⟨Expr⟩<⟨Expr⟩ | ⟨Expr⟩<=⟨Expr⟩ | ⟨Expr⟩>⟨Expr⟩ | ⟨Expr⟩>=⟨Expr⟩ | ⟨Expr⟩==⟨Expr⟩ | ⟨Expr⟩**!=**⟨Expr⟩ | ⟨Expr⟩**&&**⟨Expr⟩ | ⟨Expr⟩||⟨Expr⟩ | **!**⟨Expr⟩ | **readInteger ( )** | **readLine ( )** | **new (ident)** | **newArray (** ⟨Expr⟩,⟨Type⟩ **)**
⟨LValue⟩ ⊨ **ident** | ⟨Expr⟩**. ident** | ⟨Expr⟩**[** ⟨Expr⟩ **]**
⟨Call⟩ ⊨ **ident (**⟨Actuals⟩**)** | ⟨Expr⟩**. ident (**⟨Actuals⟩**)**
⟨Actuals⟩ ⊨ ⟨Expr⟩$^+$, | **λ**
⟨Constant⟩ ⊨ **intConstant** | **doubleConstant** | **boolConstant** | **stringConstant** | **null**

A tabela a seguir apresenta as definições regulares na forma de nome e substituição. Abrevia subexpressões comuns. O nome deve iniciar com uma letra (letras maiúsculas e minúsculas são distintas), seguida por uma sequência de letras e dígitos.

Macro	**Regex**
digit	[0-9]
letter	[a-zA-Z]
hexLetter	[a-fA-F]
id	{letter}({letter}|{digit}|[_])*
notNumber	{digit}+{id}
hex	0[xX]({digit}|{hexLetter})+
real	{digit}+\.{digit}*
exp	[eE]([+]){0,1}{digit}+
commentLine	[/][/].*

Tabela 1: Definições regulares

A tabela abaixo apresenta todos os tokens da linguagem e seus lexemas.

Token	**Lexema**
tVoid	void
tInt	int
tDouble	double
tBool	bool
tString	string
tFor	for
tWhile	while
tIf	if
tElse	else
tClass	class
tExtends	extends
tThis	this
tInterface	interface
tImplements	implements
tBreak	break
tReturn	return
tPrint	print
tReadLine	readLine
tReadLine	readLine
tNew	new
tNewArray	newArray
tPlus	+
tMinus	-
tMulti	*
tDiv	/
tMod	%
tLess	<
tLessEqual	<=
tGreater	>
tGreaterEqual	>=
tAssignment	=
tEqual	==
tDiff	!=
tAnd	&&

tOr	||
tNot	!
tSemiColon	;
tComma	,
tDot	.
tBracketLeft	[
tBracketRight	]
tParLeft	(
tParRight	)
tBraceLeft	{
tBraceRight	}
tIntConstant	{hex}|{digit}+
tDoubleConstant	{real}{exp}*
tStringConstant	\".*\"
tFalse	false
tTrue	true
tNull	null

Tabela 2: Tokens e Lexemas da linguagem

3 Manual da Linguagem

3.1 Considerações Léxicas

Durante a etapa da análise léxica da linguagem Decaf foi definido 5 tipos de tokens: identificadores, palavras reservadas, constantes, operadores e caracteres de pontuação, e comentários. Os espaços em branco, tabulações, quebras de linhas e comentários são ignorados e, no máximo, utilizados para a separação de tokens, já que, pelo menos, um deles é necessário para a separação de tokens adjacentes.

3.1.1 Palavras reservadas

As palavras reservadas da linguagem **Decaf** são palavras que não podem ser utilizadas como um identificadores, nem serem redefinidas. Abaixo segue as keywords da linguagem:

- void
- int
- double
- bool
- string
- class
- interface
- null
- this
- extends
- implements
- for
- while
- if
- else
- return
- break
- new
- newArray
- print
- readInteger
- readLine

3.1.2 Identificadores

Um identificador é uma sequência de letras, dígitos e *underlines*, começando com uma letra. O Decaf diferencia maiúsculas de minúsculas, por exemplo, *if* é uma palavra reservada, mas *IF* é um identificador e por sua vez, count e Count são dois identificadores distintos. Os identificadores podem ter no máximo 31 caracteres.

identificadores válidos
count
Count_

qtd_123
identificadores inválidos
_count
12xount
42_

3.1.3 Constantes

A linguagem Decaf possui 4 tipos de constantes: inteiras, reais, de texto e lógicas.

1. **Constantes inteiras**
   Uma constante do tipo **int** pode ser especificada na base decimal ou na base hexadecimal. Um inteiro decimal é uma sequência de dígitos decimais (0-9). Um inteiro hexadecimal deve começar com 0X ou 0x e é seguido por uma sequência de dígitos hexadecimais. Os dígitos hexadecimais incluem os dígitos decimais e as letras de A até F (maiúsculas ou minúsculas). Exemplos de números inteiros válidos: 8, 012, 0x0, 0X12aE.

   Inteiros válidos

   42

   08

   0xA56F

   0Xa56f

   0x0

   Inteiros inválidos

   x12

   0x

   12_

2. **Constantes Doubles**
   Uma constante do tipo **double** consiste em uma sequência de dígitos, um ponto, seguido ou não por outra sequência de dígitos. Um double pode possuir um expoente opcional, como por exemplo 12.2E + 2. Para um double nesta notação cientifica, o decimal mais o ponto é requerido, o sinal do expoente é opcional (se não é apresentado, + é assumido), e o 'E' pode ser tanto maiúsculo quanto minúsculo. Zeros à esquerda e no expoente são permitidos.

   Doubles válidos

   0.12

   12.

   12.E+2

   Doubles inválidos

   .12

   .12E+2

3. **Constantes Strings**
   Uma constante do tipo **String** é uma sequência de caracteres entre aspas duplas. Strings podem conter qualquer caractere exceto aspas duplas.

   Uma String deve começar e terminar em uma única linha, ou seja, não pode ser dividida em várias linhas.

   Textos válidos

   "Este é um texto válido! ;)"

   "C0mP1lAd0r3s é top!"

"Tenho 19 anos de idade."

Textos inválidos

"Esta frase está faltando as próximas aspas duplas

Esta frase não faz parte da frase de cima

4. **Constantes Lógicas**
   Uma constante do tipo **bool** possui apenas dois valores possíveis: *true* ou *false*. Como palavras-chave, essas palavras são reservadas.

3.1.4 Operadores e símbolos especiais

Os operadores e símbolos especiais utilizados pela linguagem são:

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

3.1.5 Comentários

Para comentário de linha única é iniciado por '//', e se estende até o final da linha. Comentários de várias linhas começam com '/*' e termina com o primeiro '*/' subsequente.

```
// Comentario de uma linha

/*
    Comentario que pode
    ocupar mais de uma
    linha
*/
```

4 Exemplos de Código

Segue abaixo três exemplos de código na linguagem Decaf. O primeiro algoritmo é o da busca binária.

```
int BinarySearch (int[] vector, int key, int length){
    int inf = 0;
    int sup = length-1;
    int half;
    while (inf <= sup){
        half = (inf + sup)/2;
        if (key == vector[half])
           return half;
        if (key < vector[half])
           sup = half-1;
        else
           inf = half+1;
    }
    return -1;
}
```

O segundo é a implementação do QuickSort.

```

int[] quick_sort(int[] vector, int left, int right ){
        int i; int j; int x; int y;

        i = left;
    j = right;
```

```
    x = vector[(left + right) / 2];

        while(i <= j){
        while(vector[i] < x && i < right){
            i = i + 1;
        }
        while(vector[j] > x && j > left){
            j = j - 1;
        }
        if(i <= j){
            y = vector[i];
            vector[i] = vector[j];
            vector[j] = y;
            i = i + 1;
            j = j - 1;
        }
    }

    if(j > left)
        quick_sort(vector, left, j);
    if(i < right)
        quick_sort(vector, i, right);

        return vector;
}
```

E, por fim, o terceiro é a implementação do MergeSort.

```
void merge(int[] arr, int l, int m, int r){
    int n1 = m - l + 1;
    int n2 = r - m;

    int[] L = newArray(n1, int);
    int[] R = newArray(n2, int);

    for (int i=0; i<n1; i=i+1)
        L[i] = arr[l + i];
    for (int j=0; j<n2; j=j+1)
        R[j] = arr[m + 1+ j];


    int i = 0; int j = 0;

    int k = l;
    while (i < n1 && j < n2){
        if (L[i] <= R[j]){
            arr[k] = L[i];
            i = i + 1;
        }
        else {
            arr[k] = R[j];
            j = j + 1;
        }
        k = k + 1;
    }

    while (i < n1){
        arr[k] = L[i];
        i = i + 1;
        k = k + 1;
    }
```

```
34
35      while (j < n2){
36          arr[k] = R[j];
37          j = j + 1;
38          k = k + 1;
39      }
40  }
41
42  int[] merge_sort(int[] arr, int l, int r){
43      if (l < r){
44          int m = (l+r)/2;
45
46          merge_sort(arr, l, m);
47          merge_sort(arr , m+1, r);
48
49          merge(arr, l, m, r);
50      }
51
52      return arr;
53  }
```

5 Analisador Léxico

O analisador léxico foi contruido com auxílio da ferramenta *lex*, que se trata de um programa gerador de analisadores léxicos. O analisador contruido para a linguagem é responsável por gerar a tabela de símbolos que guiará o processo de analise sintática da linguagem. Baseado na Figura 1, siga o guia de intruções abaixo para ter acesso e executar o analisador da linguagem.

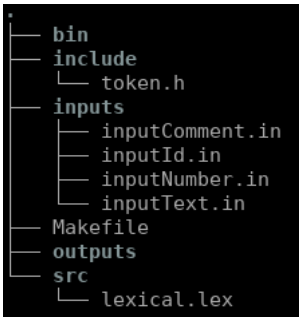

Figura 1: Árvore de diretórios

1. Abra um teminal e desloque-se até a pasta *compiler*, pasta essa que conterá o conteúdo mostrado na Figura 1;
2. execute o comando *make*. Esse será responsável por gerar o executável do programa dentro da pasta **bin**, chamado *lexical_analyzer*.
3. para executar o programa, utilizando o terminal como saída padrão, basta executar o programa *lexical_analyzer* e passar o caminho de um código fonte em decaf, ex: *.bin/lexical_analyzer* caminho_de_um_programa_em_decaf;
4. caso deseje imprimir os resultados em um arquivo, basta executar o comando acima, e o caminho de arquivo de saída, ex: *.bin/lexical_analyzer* caminho_de_um_programa_em_decaf saida_do_programa.

Vale ressaltar que todos os erros ou *warnings* mostrados pelo analisador não são inseridos no arquivo de saída, apenas são impressos na saída padrão.

6 Porcentagem de Participação

A tabela, abaixo, mostra a porcentagem de participação de cada membro da equipe no trabalho realizado.

Nome	Porcentagem
Candinho Luiz Dalla Brida Junior	33,3%
Débora Emili Costa Oliveira	33,3%
Vinícius Campos Tinoco Ribeiro	33,3%

Tabela 3: Porcentagem de participação