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)

14 de agosto de 2018

Sumário

1 Definição da Linguagem

A linguagem **Decaf** é implementada seguindo as orientações definidas em [5] e está sendo desenvolvida 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[Rib18].

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;
}
```

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
tId	{id}

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
*/
```

3.2 Subprogramas

Subprogramas são trechos de programa que realizam uma tarefa específica. Podem ser chamados pelo nome a partir do programa principal ou de trechos de outros subprogramas, até mesmo ele próprio (chamada recursiva). Na linguagem Decaf possui função, mas não possui procedimento.

Uma declaração de função inclui o nome da mesma e a assinatura de tipos associada, que inclui o tipo de retorno,e se houver, a quantidade e os tipos de parâmetros formais.

Em Decaf as funções são globais ou declaradas dentro de um escopo de classe, porém não podem ser aninhadas dentro de outras funções.

Os parâmetros formais podem ser dos tipos primitivos não-vazio, tipo de array ou tipo definido pelo usuário. O tipo de retorno da função pode ser qualquer base, matriz ou tipo definido pelo usuário. O tipo void é usado para indicar que a função não retorna nenhum valor. Se uma função tem um tipo de retorno não-vazio, qualquer declaração de retorno deve retornar um valor compatível com esse tipo. Se uma função tiver um tipo de retorno vazio, ela poderá usar apenas a instrução de retorno vazia

Sobrecarga de função não é permitida, ou seja, o uso do mesmo nome para funções com assinaturas de tipos diferentes. As funções recursivas são permitidas.

Quando uma função é invocada, os parâmetros reais são avaliados e associados aos parâmetros formais. Todos os parâmetros de Decaf e valores de retorno são passados por valor. O número de parâmetros reais em uma chamada de função deve corresponder ao número de parâmetros formais. O tipo de cada parâmetro real em uma chamada de função deve ser compatível, sendo avaliados da esquerda para a direita, com cada o parâmetro formal. Abaixo segue a exemplificação de como é declarada uma função na linguagem.

```
Type A ( type a1, ... , type an){
    ...
}
```

3.3 Classes

Declarar uma classe cria um novo nome de tipo e um escopo de classe. Uma declaração de classe é uma lista de campos, onde cada campo é uma variável ou função. O Decaf impõe o encapsulamento de objetos através de um mecanismo simples: todas as variáveis são privadas e todos os métodos são públicos. Assim, a única maneira de obter acesso ao estado do objeto é via métodos.

Todas as declarações de classe são globais, com nomes exclusivos. Um nome de campo pode ser usado no máximo uma vez dentro de um escopo de classe e podem ser declarados em qualquer ordem.

As variáveis de instância podem ser do tipo primitivo não-vazio, tipo de matriz ou tipo definido pelo usuário. O operador "."é usado para acessar os campos de um objeto. O uso de "this." é opcional ao acessar campos dentro de um método.

Objetos de uma classe são implementados como referências. Todos os objetos são alocados dinamicamente no heap usando o operador interno "new". o argumento para "new"deve ser um nome de classe (um nome de interface não é permitido)

Para invocações de método da forma *expr.metodo()*, o tipo de expr deve ser alguma classe ou interface T, o método deve ter o nome de um dos métodos de T. Acessar uma variável é dado dessa forma: *expr.var*, o tipo de expr deve ser alguma classe T, var deve ter o nome de uma das variáveis de T e esse acesso deve aparecer com o escopo da classe T ou uma de suas subclasses. Abaixo segue a exemplificação de como uma classe é declarada na linguagem.

```
1 class A {
2     ...
3 }
```

3.4 Interface

O Decaf permite que uma classe implemente uma ou mais interfaces. Uma declaração de interface consiste em uma lista de protótipos de funções sem implementação. Quando uma declaração de classe afirma que implementa uma interface, é necessário fornecer uma implementação para cada método especificado pela interface. Cada método deve corresponder ao original no tipo de retorno e nos tipos de parâmetro. Abaixo segue a exemplificação de como é declarada uma interface na linguagem.

```
1 interface A {
2     ...
3 }
```

Abaixo segue a exemplificação de como uma classe implementa uma ou mais interfaces na linguagem.

```
1 class A implements B, C{
2     ...
3 }
```

3.5 Herança

O Decaf suporta herança única, permitindo que uma classe estenda uma classe outra, adicionando campos adicionais e substituindo métodos existentes por novas definições. Uma subclasse pode substituir um método herdado (substituir por redefinição), mas a versão herdada deve corresponder ao original no tipo de retorno e nos tipos de parâmetro. O Decaf suporta upcasting automático para que um objeto da classe derivada possa ser fornecido sempre que um objeto do tipo base é esperado. Abaixo segue a exemplificação de como é realizado a herança na linguagem.

```
1 class A extends B{
2     ...
3 }
```

3.6 Estrutura de controle

As estruturas de controle da linguagem Decaf são baseadas nas versões C/Java e geralmente se comportam de maneira semelhante. A expressão nas partes de teste das instruções *if*, *while* e *for* deve ter o tipo *bool*. Uma cláusula *else* sempre se une à declaração *if* fechada mais próxima. Uma instrução *break* só pode aparecer dentro de um *while* ou *for*. Abaixo segue a exemplificação de como as estruturas de controle são utilizadas na linguagem.

```
if (expr){
    ...
}
```

```
if (expr){
    ...
} else {
    ...
}
```

```
for (expr;expr;expr;){
    ...
}
```

```
while (expr){
    ...
}
```

4 Exemplos de Código

Segue abaixo três exemplos de código na linguagem Decaf. O primeiro algoritmo é o da busca binária e foi baseado por este site [Feo16].

```
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 algoritmo abaixo é a implementação do QuickSort na linguagem Decaf e foi baseado no seguinte site [dNH12].

```

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 na linguagem Decaf, baseado no exemplo que consta em [Gee].

```
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;
    }

    while (j < n2){
```

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

5 Analisador Léxico

O analisador léxico foi construído com auxílio da ferramenta *lex*, o qual trata-se de um programa gerador de analisadores léxicos. O analisador construído para *Decaf* 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 instruções abaixo para ter acesso e executar o analisador da linguagem.

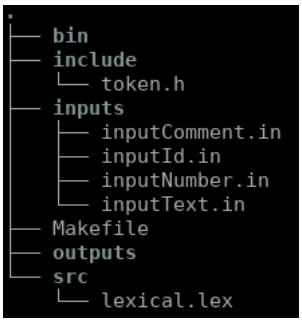

Figura 1: Árvore de diretórios

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 de erros.

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

Referências

[dNH12] Vinícius de Novais Hipólito. Algoritmo quick sort em c (quicksort). http://www.programasprontos.com/algoritmos-de-ordenacao/algortimo-quick-sort/, nov 2012. Accessed: 2018-08-10. pages 9

[Feo16] Paulo Feofiloff. Busca em vetor ordenado. https://www.ime.usp.br/ pf/algoritmos/aulas/bubi2.html, feb 2016. Accessed: 2018-08-10. pages 9

[Gee] GeeksforGeeks. Merge sort. https://www.cdn.geeksforgeeks.org/merge-sort/. Accessed: 2018-08-10. pages 10

[Rib18] Vinícius Campos Tinoco Ribeiro. decaf, aug 2018. pages 2

[5] Texas A&M University (TAMU). The decaf language. https://parasol.tamu.edu/courses/decaf/students/decafOverview.pdf. Accessed: 2018-08-10. pages 2