

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)

3 de dezembro de 2018

Sumário

1	Definição da Linguagem	2
1.1	"Hello, World!" em Decaf e linguagens na qual foi baseado	2
2	Gramática	3
3	Manual da Linguagem	5
3.1	Considerações Léxicas	5
3.1.1	Palavras reservadas	5
3.1.2	Identificadores	5
3.1.3	Constantes	6
3.1.4	Operadores e símbolos especiais	7
3.1.5	Comentários	7
3.2	Subprogramas	7
3.3	Classes	8
3.4	Interface	8
3.5	Herança	8
3.6	Estrutura de controle	9
4	Exemplos de Código	9
5	Analisador Léxico	11
6	Análise Sintática	12
6.1	Gramática LL(1)	12
6.2	Analisadores sintáticos top-down	14
6.2.1	Analisador Preditivo Recursivo	15
6.2.2	Analisador Preditivo com Tabela	15
6.3	Analisador sintático Bottom-Up	16
6.3.1	Gramática LALR	16
6.3.2	Yacc	19
6.3.3	Saída do analisador sintático	20
6.4	Árvore de Sintaxe	20
6.4.1	Árvore de Sintaxe Abstrata	21
6.4.2	Impressão da Árvore de Sintaxe Abstrata	21
7	Back-end do compilador	21
7.1	Registros de ativação	21
7.2	Geração de Código	23
7.3	Comandos	23
8	Exemplos da linguagem	24
9	Porcentagem de Participação	24
10	Conclusão	24

1 Definição da Linguagem

A linguagem **Decaf** é implementada seguindo as orientações definidas em [7] 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

```
1 void main() {  
2     print("Hello, World!");  
3 }
```

Hello World em Java

```
1 public class HelloWorld {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

Hello World em C

```
1 #include <stdio.h>  
2  
3 int main (int argc, char** argv)  
4 {  
5     printf("Hello World!\n");  
6     return (0);  
7 }
```

Hello World em C++

```
1 #include <iostream>  
2 using namespace std;  
3  
4 int main() {  
5     cout << "Hello, World!";  
6     return 0;  
7 }
```

2 Gramática

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

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

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 científica, 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.

```
1 // Comentario de uma linha
2
3 /*
4     Comentario que pode
5     ocupar mais de uma
6     linha
7 */
```

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.

```
1 Type A ( type a1, ... , type an){
2     ...
3 }
```


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.

```
1 if (expr){
2     ...
3 }
```

```
1 if (expr){
2     ...
3 } else {
4     ...
5 }
```

```
1 for (expr;expr;expr){
2     ...
3 }
```

```
1 while (expr){
2     ...
3 }
```

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].

```
1 int BinarySearch (int[] vector, int key, int length){
2     int inf;
3     int sup;
4     int half;
5
6     inf = 0;
7     sup = length-1;
8
9     while (inf <= sup){
10         half = (inf + sup)/2;
11         if (key == vector[half]){
12             return half;
13         }
14         if (key < vector[half]){
15             sup = half-1;
16         }
17         else{
18             inf = half+1;
19         }
20     }
21     return -1;
22 }
```

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

```
1 int[] quick_sort(int[] vector, int left, int right ){
2     int i;
3     int j;
```

```

4      int x;
5      int y;
6
7      i = left;
8      j = right;
9      x = vector[(left + right) / 2];
10
11     while(i <= j){
12         while(vector[i] < x && i < right){
13             i = i + 1;
14         }
15         while(vector[j] > x && j > left){
16             j = j - 1;
17         }
18         if(i <= j){
19             y = vector[i];
20             vector[i] = vector[j];
21             vector[j] = y;
22             i = i + 1;
23             j = j - 1;
24         }
25     }
26
27     if(j > left){
28         quick_sort(vector, left, j);
29     }
30     if(i < right){
31         quick_sort(vector, i, right);
32     }
33
34     return vector;
35 }

```

E, por fim, o terceiro é a implementação do MergeSort na linguagem Decaf, baseado no exemplo que consta em [Gee].

```

1 void merge(int[] vector, int l, int m, int r){
2     int n1;
3     int n2;
4     int[] l_;
5     int[] r_;
6     int i;
7     int j;
8     int k;
9
10    n1 = m - l + 1;
11    n2 = r - m;
12    l_ = newArray(n1, int);
13    r_ = newArray(n2, int);
14
15    for (i=0; i<n1; i=i+1)
16        l_[i] = vector[l + i];
17    for (i=0; i<n2; i=i+1)
18        r_[i] = vector[m + 1 + i];
19
20    i = 0;
21    j = 0;
22    k = l;
23    while (i < n1 && j < n2){
24        if (l_[i] <= r_[j]){
25            vector[k] = l_[i];

```

```

26         i = i + 1;
27     }
28     else{
29         vector[k] = r_[j];
30         j = j + 1;
31     }
32     k = k + 1;
33 }
34
35 while (i < n1){
36     vector[k] = l_[i];
37     i = i + 1;
38     k = k + 1;
39 }
40
41 while (j < n2){
42     vector[k] = r_[j];
43     j = j + 1;
44     k = k + 1;
45 }
46 }
47
48 int[] merge_sort(int[] vector, int l, int r){
49     if (l < r){
50         int m;
51
52         m = (l+r)/2;
53
54         merge_sort(vector, l, m);
55         merge_sort(vector, m+1, r);
56
57         merge(vector, l, m, r);
58     }
59
60     return vector;
61 }

```

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 análise sintática da linguagem.

Baseado na Figura 1, siga o guia de instruções abaixo para ter acesso e executar o analisador léxico da linguagem.

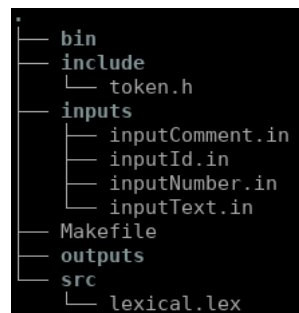


Figura 1: Árvore de diretórios

1. Abra um terminal e desloque-se até a pasta *compiler/lexical-analyzer*, 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 saída_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 Análise Sintática

O Analisador sintático também conhecido como parser tem como tarefa principal determinar se o programa de entrada representado pelo fluxo de tokens possui as sentenças válidas para a linguagem de programação. A análise sintática é a segunda etapa do processo de compilação e na maioria dos casos utiliza gramáticas livres de contexto para especificar a sintaxe de uma linguagem de programação.

6.1 Gramática LL(1)

De acordo com [ALSU08] uma gramática não pode ser LL(1) caso a mesma seja ambígua ou recursiva à esquerda. Além disso uma gramática é LL(1) se, e somente se, sempre que $A \rightarrow \alpha \mid \beta$ forem duas produções distintas de G , as seguintes condições forem verdadeiras:

1. Para um terminal a , tanto α quanto β não derivam cadeias começando por a .
2. No máximo um dos dois, α ou β , pode derivar a cadeia vazia.
3. Se $\beta \Rightarrow^* \epsilon$, então α não deriva nenhuma cadeia começando com um terminal em FOLLOW(A).
De modo semelhante, se $\alpha \Rightarrow^* \epsilon$, então β não deriva qualquer cadeia começando com um terminal em FOLLOW(A).

O primeiro "L" de LL(1), significa a varredura da entrada da esquerda para a direita. O segundo "L", a produção de uma derivação mais a esquerda e o "1", o uso de um único símbolo de entrada como *lookahead* a cada passo para tomar as decisões sintáticas.

Visto que nossa gramática original não era LL(1), foram feitos ajustes. A primeira modificação foi definir a precedência e a associatividade dos operadores, que pode ser vista na tabela abaixo:

Operadores	Precedência	Associatividade
.	1	Associativa à esquerda
[2	Não associativo
)	3	Associativa à esquerda
! -	4	Não associativo
* / %	5	Associativa à esquerda
+ -	6	Associativa à esquerda
< <= > >= == !=	7	Associativa à esquerda
&&	8	Associativa à esquerda
=	9	Associativa à direita

Tabela 3: Precedência e associatividade de operadores

Para transformar a gramática LL(1), foi preciso remover a ambiguidade do IF-ELSE aninhados. E para isso foi necessário mudar a linguagem forçando ter chaves nos blocos. Após feitas as alterações, a gramática LL(1) gerada é apresentada a seguir:

$\langle \text{Prog} \rangle$	\models	$\langle \text{Dec} \rangle \langle \text{Prog1} \rangle$
$\langle \text{Prog1} \rangle$	\models	$\lambda \mid \langle \text{Prog} \rangle$
$\langle \text{Dec} \rangle$	\models	$\langle \text{ClassDec} \rangle \mid \langle \text{InterDec} \rangle \mid \langle \text{Var} \rangle \langle \text{Dec1} \rangle \mid \mathbf{tVoid} \mathbf{tId} \langle \text{FuncDec} \rangle$
$\langle \text{Dec1} \rangle$	\models	$\mathbf{tSemiColon} \mid \langle \text{FuncDec} \rangle$
$\langle \text{Var} \rangle$	\models	$\langle \text{Type} \rangle \mathbf{tId}$
$\langle \text{Type} \rangle$	\models	$\mathbf{tInt} \langle \text{Type1} \rangle \mid \mathbf{tDouble} \langle \text{Type1} \rangle \mid \mathbf{tBool} \langle \text{Type1} \rangle \mid \mathbf{tString} \langle \text{Type1} \rangle \mid \mathbf{tUserType} \langle \text{Type1} \rangle$
$\langle \text{Type1} \rangle$	\models	$\lambda \mid \mathbf{tBracketLeft} \mathbf{tBracketRight} \langle \text{Type1} \rangle$
$\langle \text{FuncDec} \rangle$	\models	$\mathbf{tParLeft} \langle \text{Formals} \rangle \mathbf{tParRight} \langle \text{StmtBlock} \rangle$
$\langle \text{Formals} \rangle$	\models	$\lambda \mid \langle \text{Formals1} \rangle$
$\langle \text{Formals1} \rangle$	\models	$\langle \text{Var} \rangle \langle \text{Formals2} \rangle$
$\langle \text{Formals2} \rangle$	\models	$\lambda \mid \mathbf{tComma} \langle \text{Formals1} \rangle$
$\langle \text{StmtBlock} \rangle$	\models	$\mathbf{tBraceLeft} \langle \text{StmtBlock1} \rangle \mathbf{tBraceRight}$
$\langle \text{StmtBlock1} \rangle$	\models	$\langle \text{StatementList} \rangle \mid \langle \text{Var} \rangle \mathbf{tSemiColon} \langle \text{StmtBlock1} \rangle$
$\langle \text{StatementList} \rangle$	\models	$\lambda \mid \langle \text{Stmt} \rangle \langle \text{StatementList} \rangle$
$\langle \text{Stmt} \rangle$	\models	$\langle \text{ConditionStmt} \rangle \mid \langle \text{OtherStmt} \rangle$
$\langle \text{ConditionStmt} \rangle$	\models	$\mathbf{tIf} \mathbf{tParLeft} \langle \text{Expr} \rangle \mathbf{tParRight} \langle \text{StmtBlock} \rangle \langle \text{OpTail} \rangle$
$\langle \text{OpTail} \rangle$	\models	$\lambda \mid \mathbf{tElse} \langle \text{StmtBlock} \rangle$
$\langle \text{OtherStmt} \rangle$	\models	$\langle \text{WhileStmt} \rangle \mid \langle \text{ForStmt} \rangle \mid \langle \text{BreakStmt} \rangle \mid \langle \text{ReturnStmt} \rangle \mid \langle \text{PrintStmt} \rangle \mid \langle \text{StmtBlock} \rangle \mid \langle \text{ExprAssignOrCall} \rangle \mathbf{tSemiColon} \mid \mathbf{tSemiColon}$
$\langle \text{WhileStmt} \rangle$	\models	$\mathbf{tWhile} \mathbf{tParLeft} \langle \text{Expr} \rangle \mathbf{tParRight} \langle \text{Stmt} \rangle$
$\langle \text{ForStmt} \rangle$	\models	$\mathbf{tFor} \mathbf{tParLeft} \langle \text{ExprAssignOrEmpty} \rangle \mathbf{tSemiColon} \langle \text{Expr} \rangle \mathbf{tSemiColon} \langle \text{ExprAssignOrEmpty} \rangle \mathbf{tParRight} \langle \text{Stmt} \rangle$
$\langle \text{ReturnStmt} \rangle$	\models	$\mathbf{tReturn} \langle \text{ExprOrEmpty} \rangle \mathbf{tSemiColon}$
$\langle \text{BreakStmt} \rangle$	\models	$\mathbf{tBreak} \mathbf{tSemiColon}$
$\langle \text{PrintStmt} \rangle$	\models	$\mathbf{tPrint} \mathbf{tParLeft} \langle \text{Expr} \rangle \langle \text{PrintOtherStmt} \rangle$
$\langle \text{PrintOtherStmt} \rangle$	\models	$\mathbf{tParRight} \mathbf{tSemiColon} \mid \mathbf{tComma} \langle \text{Expr} \rangle \langle \text{PrintOtherStmt} \rangle$
$\langle \text{ClassDec} \rangle$	\models	$\mathbf{tClass} \mathbf{tUserType} \langle \text{ClassDec1} \rangle$
$\langle \text{ClassDec1} \rangle$	\models	$\mathbf{tExtends} \mathbf{tUserType} \langle \text{ClassDec2} \rangle \mid \langle \text{ClassDec2} \rangle$
$\langle \text{ClassDec2} \rangle$	\models	$\mathbf{tImplements} \mathbf{tUserType} \langle \text{Implements} \rangle \mathbf{tBraceLeft} \langle \text{Field} \rangle \mathbf{tBraceRight} \mid \mathbf{tBraceLeft} \langle \text{Field} \rangle \mathbf{tBraceRight}$
$\langle \text{Implements} \rangle$	\models	$\lambda \mid \mathbf{tComma} \mathbf{tUserType} \langle \text{Implements} \rangle$
$\langle \text{Field} \rangle$	\models	$\lambda \mid \langle \text{Var} \rangle \langle \text{Dec1} \rangle \langle \text{Field} \rangle \mid \mathbf{tVoid} \mathbf{tId} \langle \text{FuncDec} \rangle \langle \text{Field} \rangle$
$\langle \text{InterDec} \rangle$	\models	$\mathbf{tInterface} \mathbf{tUserType} \mathbf{tBraceLeft} \langle \text{Prototype} \rangle \mathbf{tBraceRight}$
$\langle \text{Prototype} \rangle$	\models	$\lambda \mid \langle \text{Var} \rangle \mathbf{tParLeft} \langle \text{Formals} \rangle \mathbf{tParRight} \mathbf{tSemiColon} \langle \text{Prototype} \rangle \mid \mathbf{tVoid} \mathbf{tId} \mathbf{tParLeft} \langle \text{Formals} \rangle \mathbf{tParRight} \mathbf{tSemiColon} \langle \text{Prototype} \rangle$
$\langle \text{ExprAssign} \rangle$	\models	$\langle \text{LValue} \rangle \langle \text{VariableForAssignment} \rangle \mathbf{tAssignment} \langle \text{Expr} \rangle$

$\langle \text{ExprAssignOrCall} \rangle$	$\models \langle \text{LValue} \rangle \langle \text{VariableForAssignment} \rangle \langle \text{ExprAssignOrCall1} \rangle$
$\langle \text{ExprAssignOrCall1} \rangle$	$\models \mathbf{tAssignment} \langle \text{Expr} \rangle \mid \langle \text{Call} \rangle$
$\langle \text{ExprAssignOrEmpty} \rangle$	$\models \lambda \mid \langle \text{ExprAssign} \rangle$
$\langle \text{Expr} \rangle$	$\models \langle \text{RelOp} \rangle \langle \text{LogicOp} \rangle$
$\langle \text{LogicOp} \rangle$	$\models \lambda \mid \mathbf{tOr} \langle \text{RelOp} \rangle \langle \text{LogicOp} \rangle \mid \mathbf{tAnd} \langle \text{RelOp} \rangle \langle \text{LogicOp} \rangle$
$\langle \text{RelOp} \rangle$	$\models \langle \text{PlusSubOp} \rangle \langle \text{RelOp1} \rangle$
$\langle \text{RelOp1} \rangle$	$\models \lambda \mid \mathbf{tLess} \langle \text{PlusSubOp} \rangle \langle \text{RelOp1} \rangle \mid \mathbf{tLessEqual} \langle \text{PlusSubOp} \rangle \langle \text{RelOp1} \rangle \mid$ $\mathbf{tGreater} \langle \text{PlusSubOp} \rangle \langle \text{RelOp1} \rangle \mid \mathbf{tGreaterEqual} \langle \text{PlusSubOp} \rangle \langle \text{RelOp1} \rangle \mid$ $\mathbf{tEqual} \langle \text{PlusSubOp} \rangle \langle \text{RelOp1} \rangle \mid \mathbf{tDiff} \langle \text{PlusSubOp} \rangle \langle \text{RelOp1} \rangle$
$\langle \text{PlusSubOp} \rangle$	$\models \langle \text{MulDivModOp} \rangle \langle \text{PlusSubOp1} \rangle$
$\langle \text{PlusSubOp1} \rangle$	$\models \lambda \mid \mathbf{tPlus} \langle \text{MulDivModOp} \rangle \langle \text{PlusSubOp1} \rangle \mid$ $\mathbf{tMinus} \langle \text{MulDivModOp} \rangle \langle \text{PlusSubOp1} \rangle$
$\langle \text{MulDivModOp} \rangle$	$\models \langle \text{UnaryOp} \rangle \langle \text{MulDivModOp1} \rangle$
$\langle \text{MulDivModOp1} \rangle$	$\models \lambda \mid \mathbf{tMulti} \langle \text{UnaryOp} \rangle \langle \text{MulDivModOp1} \rangle \mid \mathbf{tDiv} \langle \text{UnaryOp} \rangle \langle \text{MulDivModOp1} \rangle \mid$ $\mathbf{tMod} \langle \text{UnaryOp} \rangle \langle \text{MulDivModOp1} \rangle$
$\langle \text{UnaryOp} \rangle$	$\models \mathbf{tNot} \langle \text{UnaryOp} \rangle \mid \mathbf{tMinus} \langle \text{UnaryOp} \rangle \mid \langle \text{Term} \rangle$
$\langle \text{Term} \rangle$	$\models \langle \text{LValue} \rangle \langle \text{CallOrVariable} \rangle \mid \langle \text{Constant} \rangle \mid \mathbf{tReadInteger} \mathbf{tParLeft} \mathbf{tParRight} \mid$ $\mathbf{tReadLine} \mathbf{tParLeft} \mathbf{tParRight} \mid \mathbf{tNew} \mathbf{tParLeft} \mathbf{tId} \mathbf{tParRight} \mid$ $\mathbf{tNewArray} \mathbf{tParLeft} \langle \text{Expr} \rangle \mathbf{tComma} \langle \text{Type} \rangle \mathbf{tParRight} \mid$ $\mathbf{tParLeft} \langle \text{Expr} \rangle \mathbf{tParRight}$
$\langle \text{CallOrVariable} \rangle$	$\models \langle \text{Call} \rangle \mid \langle \text{Variable} \rangle$
$\langle \text{Call} \rangle$	$\models \mathbf{tParLeft} \langle \text{Actual} \rangle \mathbf{tParRight} \langle \text{CallVariable} \rangle$
$\langle \text{CallVariable} \rangle$	$\models \lambda \mid \langle \text{VariableNotEmpty} \rangle$
$\langle \text{VariableNotEmpty} \rangle$	$\models \mathbf{tBracketLeft} \langle \text{Expr} \rangle \mathbf{tBracketRight} \langle \text{CallAfterVariable} \rangle \mid$ $\mathbf{tDot} \langle \text{LValue} \rangle \langle \text{CallAfterVariable} \rangle$
$\langle \text{CallAfterVariable} \rangle$	$\models \lambda \mid \langle \text{VariableNotEmpty} \rangle \mid \langle \text{Call} \rangle$
$\langle \text{VariableForAssignment} \rangle$	$\models \lambda \mid \mathbf{tBracketLeft} \langle \text{Expr} \rangle \mathbf{tBracketRight} \langle \text{VariableForAssignment} \rangle \mid$ $\mathbf{tDot} \langle \text{LValue} \rangle \langle \text{VariableForAssignment} \rangle$
$\langle \text{Variable} \rangle$	$\models \lambda \mid \mathbf{tBracketLeft} \langle \text{Expr} \rangle \mathbf{tBracketRight} \langle \text{Variable1} \rangle \mid$ $\mathbf{tDot} \langle \text{LValue} \rangle \langle \text{Variable1} \rangle$
$\langle \text{Variable1} \rangle$	$\models \lambda \mid \mathbf{tBracketLeft} \langle \text{Expr} \rangle \mathbf{tBracketRight} \langle \text{Variable1} \rangle \mid$ $\mathbf{tDot} \langle \text{LValue} \rangle \langle \text{Variable1} \rangle \mid \langle \text{Call} \rangle$
$\langle \text{LValue} \rangle$	$\models \mathbf{tId} \mid \mathbf{tThis} \mathbf{tDot} \mathbf{tId}$
$\langle \text{ExprOrEmpty} \rangle$	$\models \lambda \mid \langle \text{Expr} \rangle$
$\langle \text{ExprSeq} \rangle$	$\models \lambda \mid \mathbf{tComma} \langle \text{Expr} \rangle \langle \text{ExprSeq} \rangle$
$\langle \text{Actual} \rangle$	$\models \lambda \mid \langle \text{Expr} \rangle \langle \text{ExprSeq} \rangle$
$\langle \text{Constant} \rangle$	$\models \mathbf{tIntConstant} \mid \mathbf{tDoubleConstant} \mid \mathbf{tTrue} \mid \mathbf{tFalse} \mid$ $\mathbf{tStringConstant} \mid \mathbf{tNull}$

6.2 Analisadores sintáticos top-down

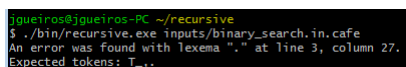
O analisador Top-Down realiza a derivação mais à esquerda de uma cadeia de entrada X a partir do símbolo inicial da gramática. A árvore gramatical desta cadeia de entrada é construída da raiz até as folhas. Em cada vértice seleciona uma produção com um símbolo não-terminal A à esquerda e constrói os vértices filhos do símbolo não terminal A com símbolos à direita nessa produção, então seleciona o vértice e continua e terminará quando todas as folhas forem símbolos terminais.

6.2.1 Analisador Preditivo Recursivo

O programa de análise de descendência recursiva, desenvolvido pelo grupo, que consiste em um conjunto de procedimentos, um para cada não-terminal. A execução começa com o procedimento para o símbolo de início, que interrompe e anuncia sucesso se o corpo do procedimento varrer toda a cadeia de entrada.

Para executar o código é preciso seguir as instruções abaixo.

1. Abra um terminal e desloque-se até a pasta *compiler/syntactic-analyzer/top-down/recursive*, pasta essa que conterá os arquivos para executar;
2. execute o comando *run.sh*. Esse será responsável por compilar o programa dentro da pasta **bin**, chamado *recursive*;
3. para executar o programa utilizando o terminal como saída padrão, basta executar o programa *recursive* e passar o caminho de um código fonte em decaf, ex: *.bin/recursive caminho_de_um_programa_em_decaf*;
4. caso o programa não apresente erros sintáticos, o programa não retorna nada. Caso contrário informa apenas o primeiro erro encontrado, a linha e a coluna. Como pode ser visto no exemplo abaixo.



```
igueiros@igueiros-PC ~/recursive
$ ./bin/recursive.exe inputs/binary_search.in.cafe
An error was found with lexema "." at line 3, column 27.
Expected tokens: T_+,.
```

Figura 2: Exemplo de erro no recursivo

6.2.2 Analisador Preditivo com Tabela

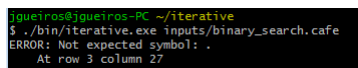
Um analisador preditivo não-recursivo pode ser construído mantendo-se uma pilha explicitamente, em vez de implicitamente, por meio de chamadas recursivas. O parser imita uma derivação mais à esquerda. Se w é a entrada que foi correspondida até agora, então a pilha contém uma sequência de símbolos gramaticais de tal forma que

$$S \xRightarrow[lm]{*} w\alpha$$

O analisador de tabela possui um buffer de entrada, uma pilha contendo uma sequência de símbolos de gramática, uma tabela de análise e um fluxo de saída. O buffer de entrada contém a string a ser analisada, seguida pelo endfile $\$$. Reutilizamos o símbolo $\$$ para marcar o final da pilha, que inicialmente contém o símbolo inicial da gramática em cima de $\$$.

Para executar o código interativo é preciso seguir as instruções abaixo.

1. Abra um terminal e desloque-se até a pasta *compiler/syntactic-analyzer/top-down/iterative*, pasta essa que conterá os arquivos para executar;
2. execute o comando *run.sh*. Esse será responsável por compilar o programa dentro da pasta **bin**, chamado *iterative*;
3. para executar o programa utilizando o terminal como saída padrão, basta executar o programa *iterative* e passar o caminho de um código fonte em decaf, ex: *.bin/iterative caminho_de_um_programa_em_decaf*;
4. caso o programa não apresente erros sintáticos, o programa não retorna nada. Caso contrário informa apenas o primeiro erro encontrado, a linha e a coluna. Como pode ser visto no exemplo abaixo.



```
lgweiras@lgweiras-PC ~/iterative
$ ./bin/iterative.exe inputs/binary_search.cafe
ERROR: Not expected symbol: .
At row 3 column 27
```

Figura 3: Exemplo de erro no interativo

6.3 Analisador sintático Bottom-Up

Uma análise Bottom-Up corresponde à construção de uma árvore de análise para uma cadeia de entrada que começa nas folhas (bottom) e trabalha em direção à raiz (topo). A análise bottom-up conhecida como análise de shift-reduce. A maior classe de gramáticas para as quais os analisadores de shift-reduce podem ser construídos, as gramáticas LR.

A fraqueza das técnicas de análise LL(k) é que elas devem prever qual produção utilizar, tendo visto apenas os primeiros k símbolos do lado direito. Uma técnica mais poderosa, a análise LR(k), é capaz de adiar a decisão até que ela tenha visto tokens de entrada correspondentes a todo o lado direito da produção em questão (e mais k tokens de entrada além).

LR(k) significa Left-to-right parse (análise da esquerda para a direita), Rightmost-derivation (derivação mais à direita), k-token lookahead (lookahead de k-token). Um analisador LR(k) usa o conteúdo de sua pilha e os próximos k-símbolos da entrada para decidir qual ação tomar. Com isso monta uma tabela com o uso de k-símbolos de lookahead.

A análise Bottom-Up necessita de uma pilha para guardar os símbolos e de um buffer de entrada para a sentença w a ser reconhecida. Possuindo 4 operações lícitas: shift, reduce, aceita e erro.

- empilha (shift):
coloca no topo da pilha o símbolo que está sendo lido e avança o cabeçote de leitura na string
- reduz (reduce):
substitui o handle no topo da pilha pelo não terminal correspondente
- aceita:
reconhece que a sentença foi gerada pela gramática
- erro:
ocorrendo erro de sintaxe, chama uma subrotina de atendimento a erro

6.3.1 Gramática LALR

Para transformar a gramática em LALR foi necessário efetuar algumas alterações na mesma. A gramática LALR é apresentada a seguir:

```
program:      decl program
            |      %empty
            ;

decl:         variableDecl
            |      functionDecl
            |      classDecl
            |      interfaceDecl
            |      error
            ;

variableDecl: variable ';'
            ;

variableDeclList: variableDecl variableDeclList
            |      %empty
            ;

variable:     type ID
            ;
```

```

type:    INT
        |    DOUBLE
        |    BOOL
        |    STRING
        |    USERTYPE
        |    type '[' ']'
        ;

functionDecl:    type ID '(' formals ')' stmtBlock
                |    VOID ID '(' formals ')' stmtBlock
                ;

formals:    variable formals1
            |    %empty
            ;

formals1:    ',' variable formals1
            |    %empty
            ;

classDecl:    CLASS USERTYPE extends implements '{' field '}'
            ;

extends:    EXTENDS USERTYPE
            |    %empty
            ;

implements:    IMPLEMENTS USERTYPE implements1
              |    %empty
              ;

implements1:    ',' USERTYPE implements1
              |    %empty
              ;

field:    variableDecl field
         |    functionDecl field
         |    %empty
         ;

interfaceDecl:    INTERFACE USERTYPE '{' prototype '}'
                 ;

prototype:    type ID '(' formals ')' ';' prototype
             |    VOID ID '(' formals ')' ';' prototype
             |    %empty
             ;

stmtBlock:    '{' variableDeclList stmtList '}'
            ;

stmt:    expr1 ';'
        |    ifStmt
        |    whileStmt
        |    forStmt
        |    breakStmt
        |    returnStmt
        |    printStmt
        |    stmtBlock

```

```

|    error
;

stmtList:  stmt stmtList
|          %empty
;

ifStmt: IF '(' expr ')' stmt
|       IF '(' expr ')' stmt ELSE stmt
;

whileStmt: WHILE '(' expr ')' stmt
;

forStmt:   FOR '(' expr1 ';' expr ';' expr1 ')' stmt
;

returnStmt: RETURN expr1 ';'
;

breakStmt:  BREAK ';'
;

printStmt:  PRINT '(' exprList ')' ';'
;

expr:  lValue '=' expr
|      constant
|      lValue
|      THIS
|      call
|      '(' expr ')'
|      expr '+' expr
|      expr '-' expr
|      expr '*' expr
|      expr '/' expr
|      expr '%' expr
|      '-' expr %prec UMINUS
|      expr L expr
|      expr LEQ expr
|      expr G expr
|      expr GEQ expr
|      expr EQ expr
|      expr NEQ expr
|      expr AND expr
|      expr OR expr
|      '!' expr
|      READINTEGER '(' ')'
|      READLINE '(' ')'
|      NEW '(' USERTYPE ')'
|      NEWARRAY '(' expr ',' type ')'
;

exprList:  expr exprList1
|          %empty
;

exprList1:  ',' expr exprList1
|           %empty
;

```

```

expr1:  expr
      |  %empty
      ;

lValue: ID
      |  expr '.' ID
      |  expr '[' expr ']'
      ;

call:   ID '(' exprList ')'
      |  expr '.' ID '(' exprList ')'
      ;

constant:  INTCONSTANT
          |  DOUBLECONSTANT
          |  TRUE
          |  FALSE
          |  STRINGCONSTANT
          |  TNULL
          ;

```

6.3.2 Yacc

O **Yacc** (“Yet another compiler-compiler”) é um gerador de analisador sintático clássico, no qual gera o analisador sintático que é parte do compilador responsável por fornecer sentido sintático a um determinado código-fonte, baseado na gramática formal.

Assim como a ferramenta Lex, o Yacc recebe como entrada um arquivo de especificação de uma gramática e gera como saída um módulo com código-fonte em C contendo uma rotina(**yyparse**) que realiza o reconhecimento de sentenças segundo essa gramática.

A utilização do Yacc se dá em conjunto com o gerador de analisador léxico Lex, onde o Yacc usa a gramática formal para analisar um entrada sintaticamente, algo que o Lex não consegue fazer utilizando expressões regulares, pois ele é limitado a simples máquinas de estado finito. Entretanto, o Yacc não consegue ler a partir de uma simples entrada de texto, ele requer uma série de tokens, que são fornecidos pelo Lex.

A especificação Yacc é dividida em três seções separadas por **%%**. São declarações do parser, regras gramaticais e rotinas do usuário.

Declarações do parser - Incluem uma lista dos símbolos terminais, não-terminais, o símbolo sentencial da gramática, a precedência dos operadores e assim por diante:

```

1  %{
2      int yylex();
3  %}
4  %token VOID INT DOUBLE
5
6  %start program
7
8  %right '='
9  %left '+' '-'
10 %left '*' '/' '%'
11 %left '.'
12
13 %%

```

Regras gramaticais - Contém as produções da gramática em BNF:

```

1  program: decl program
2          |  %empty
3          ;
4

```

```

5 decl:    variableDecl
6      |    functionDecl
7      |    classDecl
8      |    interfaceDecl
9      |    error
10     ;
11
12 %%

```

Rotinas do usuário - Contém a função principal `main()` e outras rotinas do usuário:

```

1 int main() {
2     yyparse();
3 }

```

De acordo com [AG04] o Yacc reporta conflitos de shift-reduce e reduce-reduce. Um conflito ocorre quando tanto uma operação quanto a outra seriam válidas, porém precisam ser tratadas. Um conflito de shift-reduce é uma escolha entre a operação de shift e a operação de reduce; Um conflito reduce-reduce é uma escolha de redução por duas regras diferentes. Por padrão, o Yacc resolve os conflitos de shift-reduce, optando pelo shift e os conflitos reduce-reduce, usando a regra que aparece mais cedo na gramática.

6.3.3 Saída do analisador sintático

Para executar o analisador sintático bottom-up é preciso seguir as instruções abaixo.

1. Abra um terminal e desloque-se até a pasta `compiler/syntactic-analyzer/bottom-up`, pasta essa que conterá os arquivos para executar;
2. execute o comando `run.sh`. Esse será responsável por compilar o programa dentro da pasta **bin**, chamado `parser`;
3. para executar o programa utilizando o terminal como saída padrão, basta executar o programa `parser` e passar o caminho de um código fonte em decaf, ex: `./bin/parser caminho_de_um_programa_em_decaf`;
4. caso o programa não apresente erros sintáticos, o programa retorna uma mensagem de sucesso. Como pode ser vista abaixo.
5. Caso contrário informa que foi encontrado um erro, indicando a linha e a coluna, e abaixo informa que o programa foi finalizado com a quantidade de erros encontrado. Como pode ser visto no resultado do programa abaixo.

```

1 int main(){
2     while(){
3     }
4     return 0;
5 }
6 int a = 5;

```

```

[vinihcampos@viniciuscampos bottom-up]$ ./bin/parser ../../examples/errors.cafe
An error was found in row 2 and column 12!
An error was found in row 6 and column 8!
Program finished with 2 error(s)!

```

Figura 4: Exemplo de erro no bottom-up

6.4 Árvore de Sintaxe

Um compilador deve fazer mais do que reconhecer se uma sentença pertence à linguagem de uma gramática - ela deve fazer algo útil com essa frase. As ações semânticas de um analisador podem fazer coisas úteis com as frases que são analisadas.

É possível escrever um compilador inteiro que se encaixe nas frases de ação semântica de um analisador Yacc. No entanto, tal compilador é difícil de ler e manter. E essa abordagem restringe o compilador a analisar o programa exatamente na ordem em que é analisado.

Para melhorar a modularidade, é melhor separar os problemas de sintaxe (análise) dos problemas de semântica (verificação de tipo e conversão para código de máquina). Uma maneira de fazer isso é para o analisador produzir uma árvore de análise - uma estrutura de dados que as fases posteriores do compilador podem percorrer. Tecnicamente, uma árvore de análise tem exatamente uma folha para cada símbolo da entrada e um nó interno para cada regra gramatical reduzida durante a análise.

6.4.1 Árvore de Sintaxe Abstrata

A Árvore Sintática Abstrata, ou AST, do inglês, Abstract Syntax Tree, é uma árvore n-ária onde os nodos intermediários representam símbolos não terminais, os nodos folha representam tokens presentes no programa fonte, e a raiz representa o programa corretamente analisado.

Essa árvore registra as derivações reconhecidas pelo analisador sintático, e torna mais fáceis as etapas posteriores de verificação e síntese, já que permite consultas em qualquer ordem. A árvore é abstrata, porque não precisa representar detalhadamente todas as derivações.

Tipicamente serão omitidas derivações intermediárias onde um símbolo não terminal gera somente um outro símbolo terminal, tokens que são palavras reservadas, e todos os símbolos “de sincronismo” ou identificação do código, os quais estão implícitos na estrutura reconhecida.

Os nós da árvores serão de tipos relacionados aos símbolos não terminais, ou a nós que representam operações diferentes, no caso das expressões.

O compilador precisará representar e manipular árvores de sintaxe abstrata como estruturas de dados. No nosso código em C++, essas estruturas de dados são organizadas em classes, onde cada classe é um token. O analisador Yacc, analisando a sintaxe concreta, constrói a árvore de sintaxe abstrata.

6.4.2 Impressão da Árvore de Sintaxe Abstrata

Para obter a AST basta seguir os mesmos passos descritos no item 6.3.3, com ressalva ao momento da execução, onde deve-se adicionar a flag *ast* na execução, como descrito na Figura 5.

```
[vini@campos:~/vini@campos bottom-up]$ ./bin/parser ../../examples/miniprogram.cafe ast
Program compiled successfully!
{DeclarationFunction: { id: main, type: { base: INT, size: 0 }, formal: [], StatementBlock: { DeclarationVariables: [], Statements: [{ OperatorAssignment: { leftExpression: LValueId: { id: a }, rightExpression: Call: { id: readinteger, actuals: [] }, }, { StatementPrint: { expressions: [OperatorMulti: { leftExpression: LValueId: { id: a }, rightExpression: IntConstant: { value: 10 }, }, ], }, { StatementReturn: { expression: IntConstant: { value: 0 } }, }, ] } } }
```

Figura 5: Árvore sintática abstrata

A representação escolhida para a árvore foi a estrutura de códigos **javascript**, então para ter uma melhor visualização basta acessar o site <https://www.danstools.com/javascript-beautify/> e copiar o resultado para o formatador, onde por fim será gerado uma visualização mais agradável da árvore, como demonstrado na Figura 6.

7 Back-end do compilador

A fase de análise semântica de um compilador deve traduzir a sintaxe abstrata em código de máquina abstrata. Isso pode ser feito após a verificação de tipos ou ao mesmo tempo.

Nesta etapa, por decisão de projeto em acordo com o professor da disciplina, consideraremos que os tipos estão corretos. Assim, a parte de verificação de tipos foi abstraída e não foi implementada. Com isso, o foco deste capítulo será na geração de código intermediário.

7.1 Registros de ativação

Quando lidamos com linguagens de programação modernas, normalmente trabalhamos com funções e procedimentos, que nada mais são do que uma forma de modularizar partes do código e facilitar a reutilização de código, de modo a simplificar a escrita do programa. Trabalhar com código modular

```

1 {
2   DeclarationFunction: {
3     id: main,
4     type: {
5       base: INT,
6       size: 0
7     },
8     formals: [],
9   },
10  StatementBlock: {
11    DeclarationVariables: [],
12    Statements: [{
13      OperatorAssignment: {
14        leftExpression: LValueId: {
15          id: a
16        },
17        rightExpression: Call: {
18          id: readinteger,
19          actuals: []
20        },
21      },
22    }, {
23      StatementPrint: {
24        expressions: [OperatorMulti: {
25          leftExpression: LValueId: {
26            id: a
27          },
28          rightExpression: IntConstant: {
29            value: 10
30          },
31        }, 1],
32      },
33    }, {
34      StatementReturn: {
35        expression: IntConstant: {
36          value: 0
37        },
38      },
39    }, 1],
40  },
41 },
42 }

```

Figura 6: Árvore sintática abstrata formatada

facilita a modificação e permite que diferentes programadores escrevam diferentes partes de um mesmo programa.

Além disso, em praticamente todas as linguagens modernas, uma função pode possuir variáveis locais que são criadas na entrada de uma função. E como podem ocorrer invocações da função, estas podem ocorrer ao mesmo tempo, e cada chamada possuir suas próprias instanciações de variáveis locais. Ocorre que em muitas linguagens, as variáveis locais são destruídas após o retorno da função, já que o retorno da função ocorre somente após todas as instanciações de variáveis e depois de todas as funções e procedimentos que ela chamou retornarem. As chamadas de função se comportam de maneira last-in-first-out (LIFO), assim se as variáveis são chamadas no início da função e destruídas na saída da mesma, então pode-se usar uma estrutura LIFO para armazená-las.

Porém, para linguagens que permitem funções/procedimentos aninhados, pode ser necessário manter as variáveis locais depois de uma função tiver sido retornada. A combinação de funções aninhadas e funções retornadas como resultado (ou armazenadas em variáveis) que fazem com que variáveis locais necessitem de um tempo maior de vida do que suas invocações de funções.

Deste modo, a medida que um programa executa e rotinas vão sendo executadas, a pilha de execução cresce, refletindo as chamadas. Além dos endereços de retorno, é comum para cada procedimento usar a pilha para armazenar variáveis locais e outros temporários. Por isso, associamos a cada chamada de procedimento uma área da pilha, chamada de *registro de ativação*.

Do ponto de vista de um compilador, há um monte de coisas que deve ficar na pilha relacionada à chamada/retorno do procedimento. Eles incluem endereço de devolução, parâmetros e outros vários registros. Cada procedimento tem requisitos diferentes para números de parâmetros, seu tamanho e quantos registros (quais) precisará ser salvo na pilha. Então, nós compomos um *frame de pilha* (Stack Frame) ou *registro de ativação* específico de um procedimento.

O espaço para um frame de pilha é colocado na pilha toda vez um procedimento é chamado e retirado da pilha toda vez que um retorno ocorre. Esses frames de pilha são empurrados / estourados *dinamicamente* (enquanto o programa está em execução). A pilha é usada para mais do que apenas empurrar/estourar frames de pilha. Durante a execução de um procedimento, pode haver necessidade de armazenamento de variáveis.

7.2 Geração de Código

Uma representação intermediária é um tipo de linguagem de máquina abstrata que pode expressar as operações da máquina de destino sem se comprometer com muitos detalhes específicos da máquina. Mas também é independente dos detalhes da linguagem de origem. O front-end do compilador faz análise lexical, análise analítica semântica e tradução para representação intermediária. O back-end faz a otimização da representação intermediária e da tradução para a linguagem de máquina.

A representação intermediária para a linguagem **Decaf** é feita gerando códigos para a linguagem C/C++. Onde é possível gerar código para as expressões da linguagem Decaf, os statemens, os procedimentos/funções e para declaração de classes.

Porém, a criação de código foi feita com algumas restrições. Elas são apresentadas abaixo

1. As funções só podem ser chamadas como uma instrução, ou em uma atribuição simples (e.x `variavel = funcao(params...);`);
2. Caso seja necessário usar o resultado de uma função dentro da condição do `while`, `if` ou `for`, basta armazenar o retorno em uma variável e usá-la normalmente;
3. Os métodos das classes não podem declarar variáveis com o mesmo nome de atributos da classe.

7.3 Comandos

Para ser gerado o código, inicialmente deve-se deslocar até a pasta **code-generation**, então executa-se de forma semelhante a geração da árvore abstrata, mudando a flag para **code**. Para jogar a saída em um arquivo, basta adicionar o código restante na linha de comando: **> nome_do_arquivo.cpp**.

Após isso, basta compilar o arquivo como um programa comum em C++, mais facilmente pode-se usar: **make nome_do_arquivo**. Segue abaixo um exemplo:

```

1  void main(){
2      int a;
3      int b;
4
5      a = 5;
6      b = 3;
7
8      print(a * b);
9  }
```

Figura 7: Código em decaf

A Figura 7 mostra o código na linguagem decaf.

```

[vinihcampos@vinihiuscampos code-generator]$ ./bin/parser ../../examples/miniprogram.cafe code > miniprogram.cpp
Program compiled successfully!
[vinihcampos@vinihiuscampos code-generator]$ make miniprogram
g++ miniprogram.cpp -o miniprogram
[vinihcampos@vinihiuscampos code-generator]$ ./miniprogram
15
```

Figura 8: Comandos e execução do código gerado

A Figura 8 mostra os passos necessários desde a geração até a execução do código gerado. Já a Figura 9 mostra o código gerado em C++.


```

1  #include <cstdio>
2  #include <iostream>
3  #include <stack>
4  #include <string>
5
6  using namespace std;
7
8  // Structs' definitions
9
10 // Stacks' definitions
11
12 // Returns' definitions
13
14 // Auxiliar variables
15 int pc = 0;
16 int label;
17 bool eval = false;
18 int readIntAux;
19 std::string readStringAux;
20
21 int main(){
22     int a;
23     int b;
24     a = 5;
25     b = 3;
26     cout << a*b<< endl;
27     return 0;
28     labels:{
29         switch(label){
30             default:
31                 return 0;
32         }
33     }
34 }

```

Figura 9: Código gerado

8 Exemplos da linguagem

Todos os exemplos da linguagem estão presentes na pasta *examples*, que está na raiz do projeto.

9 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	30,0%
Débora Emili Costa Oliveira	30,0%
Vinícius Campos Tinoco Ribeiro	40,0%

Tabela 4: Porcentagem de participação

10 Conclusão

Neste tópico iremos falar das nossas experiências em relação ao trabalho proposto. Este foi dividido em 5 tarefas: criação do analisador léxico, criação do analisador sintático Top-Down, criação do analisador sintático Bottom-up, criação da árvore sintática abstrata e por último a geração de código intermediário.

Na primeira tarefa, tudo ocorreu como o esperado e o nosso grupo até terminou antes do previsto. Já segunda tarefa, nós tivemos dificuldades em passar a nossa gramática para a forma LL(1). O que nos atrasou bastante, mas conseguimos entregar a tempo. Na terceira tarefa, sentimos mais facilidade em utilizar o gerador de análise sintática clássica, o *Yacc*. A tarefa seguinte foi realizada sem grandes impasses. Nesta última tarefa, ficamos com a parte de geração de código intermediária.

Não conseguimos gerar para todas especificidades da linguagem, o que já era esperado pelo tempo limitado do semestre em qual estamos, mas já estamos bem felizes pelo resultados que nós obtivemos.

Apesar de todos os problemas encontrados, o grupo como um todo, sentiu que evoluiu e aprendeu bastante na área de compiladores.

Referências

[AG04] Andrew W. Appel and Maia Ginsburg. Modern compiler implementation in c, 2004. pages 20

[ALSU08] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compiladores: princípios, técnicas e ferramentas, 2008. pages 12

[dNH12] Vinícius de Novais Hipólito. Algoritmo quick sort em c (quicksort). <http://www.programasprontos.com/algoritmos-de-ordenacao/algoritmo-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

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