# Descrição da Linguagem DECAF

Eduardo Ferreira dos Santos

24 de maio de 2017

**Abstract**

The project for the course is to write a compiler for a language called Decaf. Decaf is a simple imperative language similar to C or Pascal [1].

## 1 Considerações Léxicas

Todas as palavras reservadas em Decaf são em caixa baixa (minúsculas). As palavras reservadas e identificadores são *case sensitive*: enquanto `if` é uma palavra reservada, `IF` é um nome de variável; `foo` e `Foo` são dois nomes diferentes que se referem a duas variáveis distintas.

Algumas considerações sobre a análise léxica:

- As palavras reservadas para a linguagem são:

```
boolean break callout class continue else false for int
return true void
```

- `Program` não é uma palavra reservada, mas um identificador com significado especial, dependendo da circunstância.

- Os comentários começam com // e terminam no fim da linha.

- O espaço em branco (*white space*) pode aparecer entre quaisquer *tokens*. É definido como:

  - Um ou mais espaços;
  - *Tabs*;
  - Caracteres de quebra de página e de linha;
  - Comentários.

- Palavras reservadas e identificadores devem ser separados ou por espaço em branco, ou um *token* que não seja uma palavra reservada, ou um identificador. Por exemplo: `thisfortrue` representa um único identificador, e não três palavras reservadas distintas. Se a palavra começa com um caractere do alfabeto ou *underscore* (_), então a maior sequência de caracteres seguidos forma um *token*.

- Literais (*string literals*) são compostos por ⟨*char*⟩ entre aspas duplas. Um caractere (*char literal*) é um ⟨*char*⟩ entre aspas simples;

- Os números em Decaf são assinados em 32 bits (*32 bits signed*), ou seja, valores inteiros entre -2147483648 e 2147483647. Se uma palavra começa com `0x`, esses dois primeiros caracteres e a maior sequência

na lista `[0-9a-fA-F]` formam um número inteiro hexadecimal. Se a palavra começa com um dígito decimal (que não seja `0x`), o maior prefixo de dígitos decimais formam um inteiro decimal. Perceba que a validação do *range* será realizada depois: uma sequência longa de dígitos como `123456789123456789` ainda é reconhecida como um único *token*.

- Um tipo ⟨*char*⟩ é qualquer caractere da tabela ASCII (valores decimais entre 32 e 126, ou 40 e 176 em octal) que não seja aspas duplas ("), aspas simples ('), contrabarra (\), mais as sequências de dois caracteres para do tipo "\"" para representar aspas duplas, "\"' para representar aspas simples, "\\" para representar a contrabarra "\t" para representar o `tab` e "\n" para representar uma quebra de linha.

# 2 Gramática de Referência

Os termos léxicos descritos na seção anterior podem ser representados em termos da gramática descrita na seção. Algumas regras sobre as notações utilizadas para a descrição da gramática estão descritas na Tabela 1.

⟨*program*⟩ → **class Program** '{' ⟨*field_ decl*⟩* ⟨*method_ decl*⟩* '}'

⟨*field_ decl*⟩ → { ⟨*type*⟩ ⟨*id*⟩ | ⟨*type*⟩ ⟨*id*⟩ '[' ⟨*int_ literal*⟩ ']' }$^{+}$, ';'

| | |
|---|---|
| $\langle foo \rangle$ | foo não é um terminal, ou seja, é uma variável |
| **foo** | (em **negrito**) significa que **foo** é um terminal, ou seja, um token ou parte de um token |
| $[x]$ | zero ou uma ocorrência de $x$, ou seja, $x$ é opcional. Perceba que o uso de colchetes entre aspas ('[' ']') representa um terminal. |
| $x^*$ | uma ou mais ocorrências de $x$ |
| $x^+$, | lista de ocorrências de $x$ separadas por vírgula |
| {} | chaves grandes são utilizadas para agrupar elementos. Chaves entre aspas ('{' '}') são terminais. |
| \| | separa as alternativas (operador ou) |

Tabela 1: Meta-notação para aplicação na gramática da linguagem Decaf [1]

$\langle method\_decl \rangle \rightarrow$ { $\langle type \rangle$ | **void** } $\langle id \rangle$ '(' [ {$\langle type \rangle \langle id \rangle$}$^+$, ] ')' $\langle block \rangle$

$\langle block \rangle \rightarrow$ '{' $\langle var\_decl \rangle^*$ $\langle statement \rangle^*$ '}'

$\langle var\_decl \rangle \rightarrow \langle type \rangle \langle id \rangle^+$, ';'

$\langle type \rangle \rightarrow$ **int** | **boolean**

$\langle statement \rangle \rightarrow \langle location \rangle \langle assign\_op \rangle \langle expr \rangle$ ';'

| $\langle method\_call \rangle$ ';'

| **if** ( $\langle expr \rangle$ ) $\langle block \rangle$ [ **else** $\langle block \rangle$ ]

| **for** ( $\langle id \rangle = \langle expr \rangle$ ';' $\langle expr \rangle$ ';' $\langle block \rangle$ )

| **return** [ $\langle expr \rangle$ ] ';'

| **break** ';'

| **continue** ';'

| ⟨*block*⟩

⟨*assign_op*⟩ → =

| +=

| -=

⟨*method_call*⟩ → ⟨*method_name*⟩ ( [ ⟨*expr*⟩$^{+}$ , ] )

| **callout** ( ⟨*string_literal*⟩ [, ⟨*callout_arg*⟩$^{+}$ , ] )

⟨*method_name*⟩ → ⟨*id*⟩

⟨*location*⟩ → ⟨*id*⟩

| ⟨*id*⟩ '[' ⟨*expr*⟩ ']'

⟨*expr*⟩ → ⟨*location*⟩

| ⟨*method_call*⟩

| ⟨*literal*⟩

| ⟨*expr*⟩ ⟨*bin_op*⟩ ⟨*expr*⟩

| - ⟨*expr*⟩

| ! ⟨*expr*⟩

| ( ⟨*expr*⟩ )

$\langle callout\_arg \rangle \rightarrow \langle expr \rangle \mid \langle string\_literal \rangle$

$\langle bin\_op \rangle \rightarrow \langle arith\_op \rangle \mid \langle rel\_op \rangle \mid \langle eq\_op \rangle \mid \langle cond\_op \rangle$

$\langle arith\_op \rangle \rightarrow + \mid - \mid * \mid / \mid \%$

$\langle rel\_op \rangle \rightarrow \langle \mid / \mid \rangle \mid \langle = \mid / \mid \rangle =$

$\langle eq\_op \rangle \rightarrow == \mid !=$

$\langle cond\_op \rangle \rightarrow \&\& \mid ||$

$\langle literal \rangle \rightarrow \langle int\_literal \rangle \mid \langle char\_literal \rangle \mid \langle bool\_literal \rangle$

$\langle id \rangle \rightarrow \langle alpha \rangle \langle alpha\_num \rangle^*$

$\langle alpha\_num \rangle \rightarrow \langle alpha \rangle \mid \langle digit \rangle$

$\langle alpha \rangle \rightarrow$ a | b | ... | z | A | B |... | Z | _

$\langle digit \rangle \rightarrow 0 \mid 1 \mid ... \mid 9$

$\langle hex\_digit \rangle \rightarrow \langle digit \rangle \mid$ a | b | c | d | e | f | A | B | C | D | E | F

$\langle int\_literal \rangle \rightarrow \langle decimal\_literal \rangle \mid \langle hex\_literal \rangle$

$\langle decimal\_literal \rangle \rightarrow \langle digit \rangle \langle digit \rangle^*$

$\langle hex\_literal \rangle \rightarrow 0\text{x} \langle hex\_digit \rangle \langle hex\_digit \rangle^*$

$\langle bool\_literal \rangle \rightarrow \textbf{true} \mid \textbf{false}$

$\langle char\_literal \rangle \rightarrow \text{'}\langle char \rangle\text{'}$

$\langle string\_literal \rangle \rightarrow \text{"}\langle char \rangle^*\text{ "}$

# Referências


[1] S. Amarasinghe and M. Rinard. Computer language engineering. Disponível em `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/` Acessado em 02/08/2016, 2010.