Descrição da Linguagem DECAF

Eduardo Ferreira dos Santos

8 de junho 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*⟩)

⟨*callout_arg*⟩ → ⟨*expr*⟩ | ⟨*string_literal*⟩

⟨*bin_op*⟩ → ⟨*arith_op*⟩ | ⟨*rel_op*⟩ | ⟨*eq_op*⟩ | ⟨*cond_op*⟩

⟨*arith_op*⟩ → + | - | * | / | %

⟨*rel_op*⟩ → <| >| <= | >=

⟨*eq_op*⟩ → == | !=

⟨*cond_op*⟩ → && | ||

⟨*literal*⟩ → ⟨*int_literal*⟩ | ⟨*char_literal*⟩ | ⟨*bool_literal*⟩

⟨*id*⟩ → ⟨*alpha*⟩ ⟨*alpha_num*⟩*

⟨*alpha_num*⟩ → ⟨*alpha*⟩ | ⟨*digit*⟩

⟨*alpha*⟩ → a | b | ... | z | A | B |... | Z | _

⟨*digit*⟩ → 0 | 1 | ... | 9

⟨*hex_digit*⟩ → ⟨*digit*⟩ | a | b | c | d | e | f | A | B | C | D | E | F

⟨*int_literal*⟩ → ⟨*decimal_literal*⟩ | ⟨*hex_literal*⟩

⟨*decimal_ literal*⟩ → ⟨*digit*⟩ ⟨*digit*⟩*

⟨*hex_ literal*⟩ → 0x ⟨*hex_ digit*⟩ ⟨*hex_ digit*⟩*

⟨*bool_ literal*⟩ → **true** | **false**

⟨*char_ literal*⟩ → '⟨*char*⟩*'

⟨*string_ literal*⟩ → "⟨*char*⟩* "

2.1 Semântica

Um programa na linguagem Decaf possui uma única declaração de classe para a classe especial chamada **Program**. As declarações de classe consistem de declarações de campo e declarações de método. As declarações de campo introduzem variáveis que podem ser acessadas de maneira global por todos os métodos do programa. As declarações de método introduzem funções/procedimentos. Um programa deve conter a declaração de um método chamado **main** que não possui parâmetros de entrada. A execução de um programa em Decaf se inicia no método **main**.

2.2 Tipos

Existem dois tipos básicos de dado em Decaf: **int** e **boolean**. Existem ainda *arrays* de inteiro (`int [ N ]`) e boolean (`boolean [ N ]`).

Os *arrays* devem ser declarados no escopo global. Todos os *arrays* são unidimensionais e possuem tamanho fixo em tempo de compilação. Os *arrays* são indexados de 0 até N-1, onde N $>$ 0 é o tamanho do *array*. A notação de colchetes tradicional é utilizada para indexar os *arrays*. Já que os *arrays* possuem tamanho fixo em tempo de compilação e não pode ser declarados como parâmetros (ou variáveis locais), não existe necessidade de consultar o tamanho do *array* em Decaf.

2.3 Regras de escopo

A linguagem Decaf possui regras de escopo simples e restritivas. Todos os identificadores devem ser definidos textualmente antes de utilizados. Por exemplo:

- Uma variável deve ser declarada antes de ser utilizada;
- Um método pode ser chamado no código somente após ser declarado (métodos recursivos são permitidos).

Existem pelo menos dois escopos válidos a todo o momento em qualquer ponto do programa em Decaf: o escopo global e o escopo do método. O escopo global consiste nos nomes dos campos e métodos introduzidos na declaração da classe especial **Program**. Os escopos de método consistem nos nomes das variáveis e nos parâmetros formais introduzidos na declaração de método. Escopos locais adicionais existem em cada ⟨*block*⟩ no código; podem vir depois de de chamadas **if** ou **for**, ou inseridos em qualquer lugar onde

⟨*statement*⟩ seja legal. Um identificador introduzido no escopo do método pode mascarar um identificador do escopo global. Da mesma forma, identificadores introduzidos no escopo local mascaram identificadores definidos em escopos interiores aninhados (declarados dentro daus funções), escopos de método e escopo global.

Os nomes das variáveis definidos no escopo do método ou no escopo local podem mascarar os nomes dos métodos no escopo global. Neste caso, o identificador pode ser utilizado somente como uma variável até que ela deixe o escopo.

Nenhum identificador pode ser definido mais de uma vez no mesmo escopo. Assim, nomes de método e campos devem ser diferentes no escopo global, e o nome das variáveis locais dos parâmetros formais devem ser diferentes em cada escopo local.

2.4 Definições

A linguagem Decaf possui dois tipos de definições: variáveis escalares locais/-globais e elementos *array* globais. Definições do tipo **int** e **boolean** podem conter valores inteiros e booleanos, respectivamente. Definições do tipo `int [ N ]` e `boolean [ N ]` são utilizadas para elementos do tipo *array*. Já que os *arrays* possuem tamanho estático definido, podem ser alocados no espaço estático de dados de um programa e não precisam ser alocados na *heap*.

Cada definição é inicializada com um valor padrão quando declarada.

Os inteiros possuem como valor padrão zero, e os booleanos possuem como valor padrão **false**. As variáveis locais precisam ser inicializadas quando o programa entra no escopo declarado. Os elementos de *array* são inicializados quando o programa inicia.

2.5 Atribuições

Atribuições são permitidas apenas para valores escalares. Para os tipos **int** e **boolean**, Decaf utiliza a semântica de cópia de valor, e a atribuição $\langle location \rangle = \langle expr \rangle$ copia o valor resultante de $\langle expr \rangle$ em $\langle location \rangle$. A atribuição $\langle location \rangle$ += $\langle expr \rangle$ incrementa o valor armazenado em $\langle location \rangle$ de acordo com $\langle expr \rangle$; é valido somente para $\langle location \rangle$ e $\langle expr \rangle$ do tipo **int**. A atribuição $\langle location \rangle$ -= $\langle expr \rangle$ decrementa o valor armazenado em $\langle location \rangle$ de acordo com $\langle expr \rangle$; é valido somente para $\langle location \rangle$ e $\langle expr \rangle$ do tipo **int**.

Ambos $\langle location \rangle$ quanto $\langle expr \rangle$ em uma atribuição devem ser do mesmo tipo. Para tipos *array*, $\langle location \rangle$ e $\langle expr \rangle$ devem se referir a um único elemento do *array* que também é um valor escalar.

É possível atribuir valor a uma declaração de parâmetro formal no corpo do método. Tais atribuições afetam somente o escopo do método.

2.6 Chamadas de método e retorno

A chamada de método envolve:

1. Passar os valores dos argumentos da chamada para o corpo do método;

2. Executar o corpo do método;

3. Retornar o valor calculado pelo método para a chamada, possivelmente com o valor.

A passagem de argumentos é definida em termos de atribuição: os parâmetros formais de um método são considerados variáveis locais dos métodos e são inicializadas, por atribuição, nos valores resultantes da avaliação das expressões. As expressões são avaliadas, em termos de atribuição, da esquerda para a direita.

O corpo do método é executado de acordo com a sequência de suas chamadas.

Um método que não tem um tipo declarado como resultado só pode ser chamado, ou seja, não pode ser utilizado em uma expressão. Quando a chamada **return** é executado, o método retorna o controle para a chamada (não é permitido utilizar expressões como resultado) ou quando não há mais chamadas definidas no corpo do método (acaba o texto do método).

Um método que retorna um resultado pode ser chamado como parte de uma expressão, momento no qual o resultado da chamada é o resultado da avaliação da expressão quando a chamada **return** é alcançada. É permitido ao controle atingir o fim do texto do método que retorna um resultado.

Um método que retorna um resultado também pode ser chamado. Neste caso, o resultado é ignorado.

2.7 Fluxos de controle

if

A chamada **if** possui a semântica tradicional. Primeiro $\langle expr \rangle$ é avaliado; se o resultado for **true**, o braço **true** é executado; caso contrário, se existir um braço **else** ele é executado. Já que a linguagem Decaf exige que tanto **if** quanto **else** estejam entre chaves ({}), não existe ambiguidade ao buscar o braço **else** correspondente a um **if**.

for

A chamada **for** é similar ao loop **do** da linguagem Fortran. O $\langle id \rangle$ é a variável de índice do loop, e mascara qualquer variável do mesmo nome definida em escopo exterior, se existir. A varável de índice do loop declara um valor inteiro custo escopo está limitado ao corpo do loop. A primeira $\langle expr \rangle$ é o valor inicial da variável de índice e a segunda $\langle expr \rangle$ é o seu valor final. Cada uma dessas expressões é avaliada somente uma vez, logo antes de executar o loop pela primeira vez. Cada expressão deve retornar um valor inteiro. O corpo do loop é executado se o valor atual do índice é menor do que o valor final. Depois de uma execução do corpo do loop a variável de índice é acrescida de 1, e seu valor atual é comparado ao valor final para decidir se a próxima iteração deve ser executada.

2.8 Expressões

As expressões seguem as regras normais de cálculo. Na ausência de outras restrições, operadores de mesma precedência são avaliados da esquerda para a direita. Parênteses podem ser utilizados para alterar a precedência normal.

Uma expressão de definição é calculada de acordo com o valor contido em sua definição.

Expressões de chamada de método serão discutidas na seção referente a *Chamadas de Método e Retorno*. Operações de *array* foram discutidas na seção de Tipos. Expressões relacionadas a operações de E/S são discutidas na seção *Chamadas de biblioteca (callouts)*.

Inteiros literais são calculados para o seu valor inteiro. Caracteres literais são calculados de acordo com seus valores inteiros na tabela ASCII. Ex.: `'A'` representa o inteiro 65. (O tipo de um caractere literal é **int**).

Os operadores aritméticos ($\langle arith\_op \rangle$ e o operador unário negativo) têm sua precedência e significado normais, assim como os operadores relacionais ($\langle rel\_op \rangle$). % computa o resto de uma divisão.

Os operadores relacionais são utilizados para comparar expressões inteiras. Os operadores de igualdade == e != são definidos somente para tipos **int** e **boolean**. Podem ser utilizados para comparar quaisquer expressões do mesmo tipo. (== é igual e != é diferente)

O resultado de um operador relacional ou de igualdade tem o tipo **boolean**.

Os conectivos lógicos booleanos && e || são interpretados utilizando ava-

Operadores	*Comentários*
-	operador unário negativo
!	não lógico
/ %	multiplicação, divisão, resto
+ -	adição, subtração
< <= >= >	relacional
== !=	igualdade
&&	and condicional
||	or condicional

Tabela 2: Precedência de operadores, da menor para a maior [1]

liação rápida em Java. Os efeitos secundários do segundo operando não são executados e o resultado do primeiro operando determina o valor de toda a expressão, ou seja, se o resultado é falso para && ou verdadeiro para ||.

A precedência de operadores está definida na Tabela 2 e não se reflete na gramática de referência.

2.9 Chamadas de biblioteca (callouts)

A linguagem Decaf inclui um método primitivo para chamar funções disponíveis no ambiente de execução, tais como a biblioteca C padrão ou funções definidas pelo usuário.

O método primitivo para chamar funções é:

> **int callout** ($\langle string\_literal \rangle$, [$\langle callout\_arg \rangle^+$,]) – a função definida na primeira *string* literal é chamada com os argumentos seguintes que são fornecidos. Expressões do tipo booleano ou inteiro são fornecidas como inteiros; *strings* ou expressões do tipo

array são fornecidas como ponteiros. O valor de retorno da função é devolvido como um inteiro. O usuário que executa a chamada de `callout` é responsável por garantir que os argumentos fornecidos estão de acordo com a assinatura da função, e o valor de retorno só é utilizado se a respectiva função retorna um valor do tipo apropriado. Argumentos são fornecidos para a função na convenção padrão de chamadas do sistema.

Em adição ao acesso à biblioteca padrão da linguagem C utlizando **callout**, uma função de E/S pode ser escrita em C ou qualquer outra linguagem, compilada utilizando as ferramentas padrão, conectada no momento da execução (*linker*) e acessada pelo mecanismo de **callout**.

3 Regras Semânticas

O conjunto de regras a seguir adiciona algumas restrições ao conjunto de programas válidos em Decaf, além daquelas fornecidas pela Gramática. Um programa que seja gramaticamente bem formado e não viole nenhuma das regras a seguir é chamado de *legal*. Um compilador robusto deve checar explicitamente cada uma dessas regras e gerar mensagens de erro para cada uma das violações encontradas, mas não vai gerar nenhuma mensagem para um programa legal.

1. Nenhum identificador é declarado duas vezes no mesmo escopo.

2. Nenhum identificador pode ser utilizado antes de ser declarado.

3. O programa contém uma definição do método chamado **main** que não possui nenhum parâmetro. Perceba que, como execução se inicia no método **main**, quaisquer métodos definidos após **main** nunca serão executados.

4. O ⟨*int_literal*⟩ numa declaração de *array* deve ser maior que zero.

5. O número de argumentos e seus tipos em uma chamada de método devem ser iguais ao número de argumentos e seus tipos na definição, ou seja, as assinaturas devem ser idênticas.

6. Se uma chamada de método é utilizada em uma expressão o método deve retornar um resultado.

7. Uma declaração **return** não deve retornar nenhum valor a não ser que o corpo do método defina a necessidade de retonar um valor.

8. A expressão em uma definição de **return** deve ter o mesmo tipo do resultado declarado na definição do método.

9. Um ⟨*id*⟩ utilizado como um ⟨*location*⟩ define uma variável global/local ou parâmetro formal.

10. Para todas as definições no formato ⟨*id*⟩ [⟨*expr*⟩]:

    (a) ⟨*id*⟩ deve ser uma variável do tipo **array**, e;

(b) O tipo de ⟨*expr*⟩ deve ser **int**.

11. A ⟨*expr*⟩ em um **if** deve ser do tipo **boolean**.

12. Os operandos de ⟨*arith_op*⟩ e ⟨*rel_op*⟩ devem ser do tipo **int**.

13. Os operandos de ⟨*eq_op*⟩ devem ser do mesmo tipo: **int** ou **boolean**.

14. Os operandos de ⟨*cond_op*⟩ e o operando do não lógico (**!**) devem ser do tipo **boolean**.

15. Em uma atribuição do tipo ⟨*location*⟩ = ⟨*expr*⟩, ⟨*location*⟩ e ⟨*expr*⟩ devem ser do mesmo tipo.

16. Em uma atribuição de acréscimo/decréscimo do tipo ⟨*location*⟩ += ⟨*expr*⟩ e ⟨*location*⟩ -= ⟨*expr*⟩, ambos ⟨*location*⟩ e ⟨*expr*⟩ devem ser do tipo **int**.

17. A ⟨*expr*⟩ inicial e a ⟨*expr*⟩ final em um **for** devem ser do tipo **int**.

18. Todas as definições **break** e **continue** devem estar contidas no corpo de um **for**.

4 Validação em tempo de execução

Em adição às restrições descritas nas seções anteriores, que estão diretamente ligadas à fase de análise semântica do compilador, as seguintes restrições são reforçadas de maneira dinâmica: o gerador de código do compilador deve

gerar código para realizar as checagens; violações são descobertas em tempo de execução.

1. O valor atribuído a um *array* deve estar dentro do limite.

2. O controle não deve terminar de executar um método que tenha declarado a necessidade de retornar um resultado.

Quando acontece um erro em tempo de execução, uma mensagem de erro apropriada é enviada ao terminal e o programa é finalizado. Tais mensagens de erro devem auxiliar o programador a tentar encontrar o problema no código-fonte.

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.