

Processamento de linguagens em Haskell (exemplo de aplicação)

17 de novembro de 2021

Formalização de linguagens

Sintaxe regras de **formação** de expressões, frases, etc.

Semântica definição do **significado** desses fragmentos

A sintaxe é formalizada usando **gramáticas livres de contexto**.

A semântica pode ser formalizada de várias formas; vamos usar **semânticas operacionais**.

Gramáticas livres de contexto

Σ conjunto de **símbolos terminais**.

V conjunto de **símbolos não-terminais**.

P conjunto de **produções** da forma

$$\langle \text{n\~ao-terminal} \rangle ::= \langle \text{alt} \rangle_1 \mid \dots \mid \langle \text{alt} \rangle_n$$

em que $\langle \text{alt} \rangle_i$ são sequências de terminais ou não-terminais.

$S \in V$ **símbolo não-terminal inicial**

Linguagem sequências de símbolos terminais geradas a partir de S usando as produções.

Exemplo: expressões aritméticas

Terminais $\Sigma = \{0, 1, \dots, 9, +, *, (,)\}$

Não-terminais $V = \{E, N, A\}$

Símbolo inicial E

Produções

$$E ::= E + E \mid E * E \mid (E) \mid N$$
$$N ::= AN \mid A$$
$$A ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

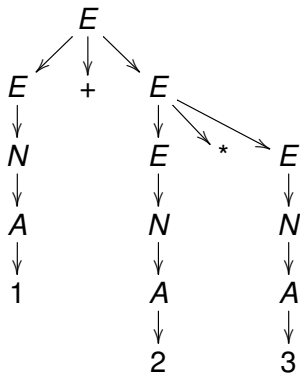
Derivações

$$\begin{aligned} E &\rightarrow E + E \rightarrow E + E * E \rightarrow N + E * E \rightarrow A + E * E \\ &\rightarrow 1 + E * E \rightarrow 1 + N * E \rightarrow 1 + A * E \rightarrow 1 + 2 * E \\ &\rightarrow 1 + 2 * N \rightarrow 1 + 2 * A \rightarrow \underbrace{1 + 2 * 3}_{\text{terminais}} \end{aligned}$$

Logo: “1+2*3” é uma **palavra** da linguagem de expressões.

Árvores de derivação

Podemos representar a derivação de uma palavra por uma árvore:



Exercício: encontrar *outra* derivação que resulta numa *árvore diferente*.¹

¹Esta gramática é *ambígua*.

Sintaxe abstracta

A **sintaxe abstracta** representa a estrutura da árvore de derivação omitindo informação desnecessária, e.g.:

- decomposição de números em algarismos
- parêntesis

Definindo um tipo de dados em Haskell:

```
data Expr = Add Expr Expr
          | Mult Expr Expr
          | Num Int
          deriving (Eq, Show)
```

Sintaxe concreta vs. abstracta

$1+2*3$

Add (Num 1) (Mult (Num 2) (Num 3))

Mult (Add (Num 1) (Num 2)) (Num 3)

$(1+2)*3$

Mult (Add (Num 1) (Num 2)) (Num 3)

$1+2+3$

Add (Add (Num 1) (Num 2)) (Num 3)

Add (Num 1) (Add (Num 2) (Num 3))

Semântica operacional

- Uma **relação** \Rightarrow entre **expressões** e **valores** (inteiros)
- Definida indutivamente sobre termos da sintaxe abstracta
- Regras de inferência da forma

$$\frac{A_1 \quad A_2 \quad \dots \quad A_n}{B}$$

A_1, A_2, \dots, A_n hipóteses

B conclusão

axioma quando $n = 0$

leitura lógica: se A_1, \dots, A_n então B

leitura computacional: para obter B , efectuar A_1, \dots, A_n .

Regras de inferência

$$\frac{}{\text{Num } n \Rightarrow n}$$

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2}{\text{Add } e_1 \ e_2 \Rightarrow n_1 + n_2}$$

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2}{\text{Mult } e_1 \ e_2 \Rightarrow n_1 \times n_2}$$

Exemplo duma derivação

$$\frac{\frac{\text{Num 1} \Rightarrow 1 \quad \text{Num 2} \Rightarrow 2}{\text{Add (Num 1) (Num 2)} \Rightarrow 3} \quad \text{Num 3} \Rightarrow 3}{\text{Mul (Add (Num 1) (Num 2)) (Num 3)} \Rightarrow 9}$$

Interpretador em Haskell

Podemos implementar \Rightarrow como uma função recursiva:

```
eval :: Expr -> Int
eval (Num n)    = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mult e1 e2) = eval e1 * eval e2
```

Exemplo:

```
eval (Mult (Add (Num 1) (Num 2)) (Num 3))
= eval (Add (Num 1) (Num 2)) * eval (Num 3)
= (eval (Num 1) + eval (Num 2)) * 3
= (1 + 2) * 3
= 9
```

Correção

O interpretador implementa correctamente a semântica operacional:

$$e \Rightarrow n \text{ se e só se } \text{eval } e = n$$

Prova: por *indução estrutural* sobre e .

Compilação de expressões

- Vamos definir uma *máquina virtual* para a linguagem das expressões aritméticas
- Traduzimos cada expressão numa *sequência de operações*
- Eliminamos a recursão: os valores intermédios passam a ser explícitos
- Para executar o código virtual já não necessitamos da árvore sintática

Uma máquina de pilha

Configuração da máquina abstracta:

pilha: uma sequência de inteiros;

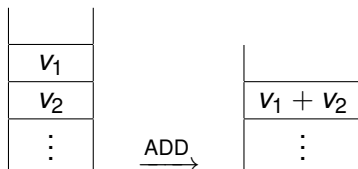
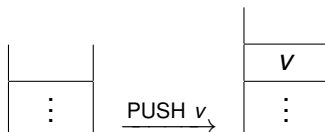
programa: uma sequência de *instruções*

Instruções da máquina abstracta:

PUSH n coloca um valor no topo da pilha

ADD } operações aritméticas
 } (sobre valores na pilha)
MUL }

Operações sobre a pilha



Analogamente para a instrução MUL.

Definições da máquina

```
-- instruções da máquina virtual
data Instr = PUSH Int
           | ADD
           | MUL
           deriving (Eq, Show)

-- a pilha é uma lista de valores
type Stack = [Int]

-- o código é uma lista de instruções
type Code = [Instr]

-- a configuração da máquina
type State = (Stack, Code)
```

Compilador de expressões

```
compile :: Expr -> Code
compile (Num n) = [PUSH n]
compile (Add e1 e2)
    = compile e1 ++ compile e2 ++ [ADD]
compile (Mul e1 e2)
    = compile e1 ++ compile e2 ++ [MUL]
```

- Traduz uma expressão numa sequência de instruções
- Análogo a *eval*, mas gera instruções em vez valores

Invariante

A execução de `compile e` acrescenta o `valor de e` ao topo da pilha.

Exemplo

```
> compile (Mult (Add (Num 1) (Num 2)) (Num 3))  
[PUSH 1, PUSH 2, ADD, PUSH 3, MUL]
```

Função de transição

Implementa a transição associada a cada instrução da máquina abstracta:

```
exec :: State -> State
exec (stack, PUSH v:code) = (v:stack, code)
exec (v1:v2:stack, ADD:code)
    = ((v1+v2):stack, code)
exec (v1:v2:stack, MUL:code)
    = ((v1*v2):stack, code)
```

Interpretador de código virtual

Iterar a função de transição até esgotar as instruções; o resultado é o **topo da pilha**.

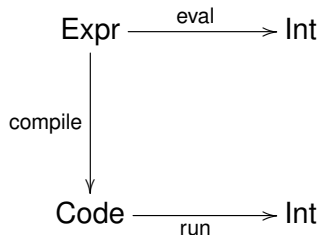
```
runState :: State -> Int
runState (v:_, []) = v
runState s          = runState (exec s)
```

```
run :: Code -> Int
run c = runState ([], c)
```

Exemplo de execução:

```
> run (compile (Mult (Add (Num 1) (Num 2)) (Num 3)))
9
```

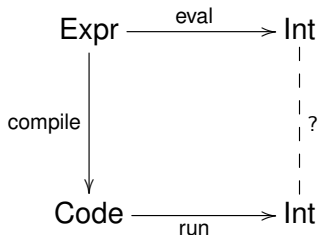
Dois processos para calcular expressões



Correção da compilação

$\forall e. \text{eval } e = \text{run } (\text{compile } e)$

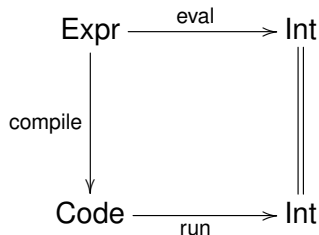
Dois processos para calcular expressões



Correção da compilação

$\forall e. \text{eval } e = \text{run } (\text{compile } e)$

Dois processos para calcular expressões



Correção da compilação

$$\forall e. \text{eval } e = \text{run } (\text{compile } e)$$

Conclusão

- Dois interpretadores em Haskell:
 - `eval`
 - um interpretador de **expressões**
 - especificação de alto-nível
 - `run`
 - interpretador de **código virtual**
 - mais perto de uma máquina real
- Correção da compilação: os dois interpretadores produzem o mesmo resultado para qualquer expressão