Interpretação e Compilação de Linguagens de Programação
Sintaxe e Semântica

28 de Fevereiro de 2013

Nesta aula apresentam-se dois dos aspetos fundamentais das linguagens de programação, sintaxe e semântica. E principalmente a representação de um programa como um tipo de dados indutivo, correspondendo à forma das linguagens de programação estruturadas. Como exemplo, é apresentada a sintaxe e semântica das linguagem das expressões aritméticas (CALC).

1 Sintaxe e Semântica

A sintaxe e a semântica são dois dos aspetos mais importantes da definição de uma linguagem de programação. Embora a semântica tenha uma presença maior nesta unidade curricular, a sintaxe é igualmente importante e objeto de muitas evoluções recentes.

A sintaxe define a forma que os programas de uma linguagem têm e determina não só a facilidade ou dificuldade com que se exprimem os conceitos, mas também pode limitar em muito a expressividade de uma linguagem (ao proibir à partida certas construções). Neste caso teremos em conta apenas programas escritos em texto, mas muitas das noções aplicam-se igualmente a outras formas de expressão (diagramáticas ou visuais).

A semântica de uma linguagem define o significado dos programas de uma linguagem de programação. Este noção genérica de significado pode ser instanciado de várias maneiras, e se para uma linguagem se define uma só sintaxe, podem ser definidas várias semânticas sem que sejam conflituantes. A definição de um interpretador, ou de um compilador ou de uma qualquer verificação para uma linguagem são instâncias diferentes de uma semântica para uma linguagem, onde os significados são de natureza diferente.

Os aspetos de sintaxe são fundamentais na definição de uma linguagem pois deles também depende o significado dos programas. Um exemplo clássico de âmbiguidade relativa a aspetos sintáticos relaciona-se com a construção *if-then-else* quando o ramo *else* é opcional. Utilizando a definição da Figura 1 não é claro qual o resultado da expressão `f(-1)`. Sintaticamente, é preciso convencionar a qual das expressões *if* pertence o único ramo else. Normalmente convenciona-se que é o *if* mais próximo, o que neste caso vai contra a indentação do exemplo. No exemplo da Figura 2, podemos ver o nome `name` utilizado de várias maneiras e contextos. Em ruby o identificador `@name` é sintaticamente diferente do identificador `name`, o primeiro denota um campo de um objeto, e o segundo denota

```
int f(int x) {
  if ( x > 0 )
    if ( x < 10 )
      return x;
  else return 10;
  return 0;
}
```

Figura 1: Âmbiguidade sintática em C

```
class Hello
  def init(name)
    @name = name
  end

  def hello
    puts "Hello #{@name}!"
  end
end
```

Figura 2: Aspeto sintático da linguagem Ruby

uma variável local, a utilização de uma expressão no meio de uma string desde que dentro de um contexto `#{...}` deve ser avalidada no contexto corrente.

Estes exemplos justificam a necessidade de haver uma definição precisa da sintaxe de uma linguagem, para que todas as ambiguidades sejam eliminadas, por desenho ou por convenção. A forma concreta como se representam os programas de uma linguagem, as palavras reservadas, os separadores, símbolos envolventes de blocos, e prioridade de operadores, define o que se chama *sintaxe concreta da linguagem*. Por contraste com esta definição, chama-se sintaxe abstrata à representação interna que os interpretadores e compiladores utilizam para denotar os programas. Esta representação é agnóstica dos aspetos concretos e retém apenas as características fundamentais.

2 Sintaxe Concreta

A sintaxe concreta de uma linguagem é definida através de um conjunto de artefactos, nomeadamente uma definição lexicográfica (que palavras - reservadas ou compostas - são reconhecidas como sendo átomos da linguagem) e uma definição gramatical (que regras de formação de frases são admissíveis para formar programas e subprogramas).

(Leituras extra sobre *lexing* e *parsing* em [AP02])

3 Sintaxe Abstrata

A sintaxe abstrata de uma linguagem define uma estrutura de dados manipulável por um algoritmo. A estruturação introduzida por Dijsktra no desenho de linguagens [**?**], permitiu a definição de um programa como um tipo de dados indutivo, ou seja, definido resursiva e composicionalmente a partir de progra-

```
/* ListInt.h */

typedef struct ListInt ListInt;

ListInt* nil();
ListInt* cons(int elem, ListInt *tail);
```

Figura 3: Interface do módulo `ListInt` em C

mas mais pequenos. A essa definição indutiva corresponde também a definição composicional e recursiva de funções e algoritmos para representar a semântica de programas. Interpretadores, compiladores e verificadores de programas são exemplos destas funções e algoritmos.

3.1 Tipos de dados indutivos

Numa definição indutiva para um tipo de dados como uma lista de elementos de um determinado tipo T, utilizam-se regras de construção como na seguinte definição:

Definition 3.1 (List)**.**
Uma lista de elementos de tipo T é definida indutivamente por:

1. ***nil** é uma lista (a lista vazia)*
2. *se x é um valor de tipo T e L é uma lista de elementos de tipo T, então **cons**(x, L) é também uma lista de elementos de tipo T.*
3. *não existem mais listas de elementos de tipo T, exceto as construídas de acordo com as regras 1 e 2.*

Esta definição garante que qualquer lista de elementos de tipo T é construída exactamente desta maneira.

Note-se que uma definição indutiva de um tipo de dados tem várias componentes, neste caso: um nome para o tipo e um conjunto de construtores. Para o tipo lista de elementos de tipo inteiro, podemos definir o nome do tipo, *ListInt*, e utilizar os construtores com os tipos:

1. **nil** : $() \rightarrow$ *ListInt*
2. **cons** : *Integer* $\times$ *ListInt* $\rightarrow$ *ListInt*

o que em corresponde, por exemplo, a uma declaração de um módulo em C com o interface descrito pela Figura 3, ou a um tipo de dados ou módulo em OCaml, Figura 4, ou através da implementação de classes em Java com um determinado interface, Figura 5.

A definição indutiva de tipos de dados permite a definição direta de algoritmos indutivos, para exprimir significados ou propriedades.

```
module type List = Sig
  type intList
  val nil:intList
  val cons: int -> intList -> intList
end
```

Figura 4: Interface do módulo `ListInt` em ML

```
interface ListInt {...};

class ListFactory {
  static ListInt nil();
  static ListInt cons(int elem, ListInt tail);
};
```

Figura 5: Interface do módulo `ListInt` em Java

4 Árvores sintáticas

A partir da árvore de derivação de uma gramática para um dado programa, é construir um valor que representa esse programa em memória. Dada uma gramática é possível definir um tipo de dados indutivo capaz de caracterizar todos os programas dessa linguagem.

A transformação de um texto escrito numa linguagem de programação num valor do tipo de dados que a representa, pode ser feita por uma ferramenta de análise lexical e gramatical. Uma linguagem como a das expressões aritméticas composta pelos numeros inteiros e os operadores $+$, $-$, $*$ e $/$, pode ser representada num tipo de dados com o nome $CALC$ e com os construtores:

1. **num** : $\mathit{Integer} \to CALC$
2. **add** : $CALC \times CALC \to CALC$
3. **sub** : $CALC \times CALC \to CALC$
4. **mul** : $CALC \times CALC \to CALC$
5. **div** : $CALC \times CALC \to CALC$

Em OCaml corresponde à definição de um tipo soma (indutivo) como na Figura 6, ou a um conjunto de interfaces e classes Java como na Figura 7.

Estes construtores abstraem aspetos da sua representação concreta como por exemplo, o uso de parentesis, a prioridade dos operadores entre si, ou mesmo as diferentes representações de literais.

5 Semântica

Depois de sabermos representar um programa como um tipo de dados, definimos o conjunto de todos os programas de uma determinada linguagem (e.g. $PROG$), e sabemos manipular os seus valores através de algoritmos. O passo seguinte é atribuir significado a esses programas.

```
type calc =
    Num of int
  | Add of calc * calc
  | Sub of calc * calc
  | Mul of calc * calc
  | Div of calc * calc
```

Figura 6: Tipo de dados OCaml dos programas da linguagem CALC

```
interface ASTNode {...}

class ASTAdd {
  ASTNode left, right;
}

class ASTSub {
  ASTNode left, right;
}

class ASTMul {
  ASTNode left, right;
}

class ASTDiv {
  ASTNode left, right;
}
```

Figura 7: Tipo de dados Java dos programas da linguagem CALC

Em geral, a semântica de uma linguagem pode ser caracterizada por uma função I que atribui um significado (ou denotação) a cada programa (ou fragmento de programa) sintacticamente correcto. I é uma função de domínio o conjunto dos programas ($PROG$) e contra-domínio o conjunto das denotações possíveis (e.g. os números inteiros, $\mathbb{Z}$). Outros conjuntos de denotações são possíveis, definindo assim funções de interpretação diferentes. Por exemplo, um sistema de tipos é uma função de interpretação que a um dado programa atribui uma denotação no conjunto dos tipos. Dessa maneira define o conjunto dos programas bem tipificados dessa linguagem. Uma função cujo contradomínio são os programas numa linguagem intermédia são programas numa linguagem máquina é um compilador.

6 Semântica operacional

Denomina-se por semântica operacional estrutural uma definição da função semântica para uma determinada linguagem que se baseia na estrutura das expressões da linguagem. Nesta técnica, o significado de uma dada expressão ou programa é obtido a partir dos significados das suas sub-expressões ou sub-programas.

Definimos a semântica operacional estrutural da linguagem CALC como sendo a semântica esperada para as expressões aritméticas, em que a denotação

```
let rec eval e =
  match e with
    Number n -> n
  | Add(l,r) -> (eval l)+(eval r)
  | Sub(l,r) -> (eval l)-(eval r)
  | Mul(l,r) -> (eval l)*(eval r)
  | Div(l,r) -> (eval l)/(eval r)
```

Figura 8: Função de avaliação da linguagem CALC

de uma expressão é o seu valor inteiro.

A definição da semântica operacional da linguagem *CALC* é feita através de uma função I

$$I : CALC \rightarrow Integer$$

onde *CALC* é o conjunto de todos os programas sintaticamente válidos e *Integer* o conjunto das denotações possíveis para uma expressão aritmética (os números inteiros). Esta função é definida indutivamente no tipo de dados *CALC*, isto é por casos, como no caso do código da Figura 8.

Esta é uma definição dita composicional, o que significa que para acrescentar uma nova construção à linguagem é somente necessário definir a semântica dessa mesma construção e não é necessário cobrir casos especiais das construções existentes.

Referências

[AP02] A.W. Appel and J. Palsberg. *Modern Compiler Implementation in Java.* Cambridge University Press, 2002.

A Haskell code

```
module Calc where

{- Definition of the Abstract Syntax of the language CALCI -}

data CALC =
    Num Int
  | Add CALC CALC
  | Sub CALC CALC
  | Mul CALC CALC
  | Div CALC CALC
  deriving Show

{- Examples -}

a = (Num 1)

b = (Add (Num 1) (Sub (Num 3) (Num 2)))
```

```
 ------------------------------------------------------------------------------
{- Semantics                                                                  -}
 ------------------------------------------------------------------------------

eval :: CALC -> Int
eval (Num n) = n
eval (Add e e') = (eval e) + (eval e')
eval (Sub e e') = (eval e) - (eval e')
eval (Mul e e') = (eval e) * (eval e')
eval (Div e e') = div (eval e) (eval e')
-- division by zero causes an exception to occur

 ------------------------------------------------------------------------------
{- Abstract syntax of results                                                 -}
 ------------------------------------------------------------------------------

data Result =
       Value Int
     | Error
     deriving Show

 ------------------------------------------------------------------------------
{- Semantics with Result (Int or Error)                                       -}
 ------------------------------------------------------------------------------

evalCase :: CALC -> Result

evalCase (Num n) = Value n

evalCase (Add e e') =
         case (evalCase e, evalCase e') of
              (Value n, Value n') -> Value (n+n')
              (_ , _) -> Error

evalCase (Sub e e') =
         case (evalCase e, evalCase e') of
              (Value n, Value n') -> Value (n-n')
              (_ , _) -> Error

evalCase (Mul e e') =
         case (evalCase e, evalCase e') of
              (Value n, Value n') -> Value (n*n')
              (_ , _) -> Error

evalCase (Div e e') =
         case (evalCase e, evalCase e') of
              (Value n, Value 0) -> Error
              (Value n, Value n') -> Value (div n n')
              (_ , _) -> Error
```

```
 ------------------------------------------------------------------------------
{- Semantics with Result (Int or Error) using Maybe Monad                     -}
 ------------------------------------------------------------------------------

evalErr :: CALC -> Maybe Int

evalErr (Num n) = Just n

evalErr (Add e e') = do
        l <- evalErr e
        r <- evalErr e'
        return (l+r)

evalErr (Sub e e') = do
        l <- evalErr e
        r <- evalErr e'
        return (l-r)

evalErr (Mul e e') = do
        l <- evalErr e
        r <- evalErr e'
        return (l*r)

evalErr (Div e e') = do
        l <- evalErr e
        r <- evalErr e'
        if r == 0 then Nothing else Just (div l r)

-- evalErr (Num 1)
-- evalErr (Add (Num 1) (Sub (Num 3) (Num 2)))

 ------------------------------------------------------------------------------
{- Definition of a specific Error Monad                                       -}
 ------------------------------------------------------------------------------

data ErrorMonad a =
       Result a
     | Wrong
     deriving Show

instance Monad ErrorMonad
  where
    -- (>>=) :: ErrorMonad a -> (a -> ErrorMonad b) -> ErrorMonad b
    p >>= k  =  case p of
                  Result a -> k a
                  Wrong -> Wrong

    -- return :: a -> ErrorMonad a
```

```
    return a = Result a

raise_error :: ErrorMonad a
raise_error = Wrong

 ------------------------------------------------------------------------------
{- Semantics using the ErrorMonad                                             -}
 ------------------------------------------------------------------------------

evalWrong :: CALC -> ErrorMonad Int

evalWrong (Num n) = return n

evalWrong (Add e e') = do
        l <- evalWrong e
        r <- evalWrong e'
        return (l+r)

evalWrong (Sub e e') = do
        l <- evalWrong e
        r <- evalWrong e'
        return (l-r)

evalWrong (Mul e e') = do
        l <- evalWrong e
        r <- evalWrong e'
        return (l*r)

evalWrong (Div e e') = do
        l <- evalWrong e
        r <- evalWrong e'
        if r == 0 then raise_error else return (div l r)
```