Interpretação e Compilação de Linguagens de Programação
Ligação e âmbito

8 de Março de 2013

Nesta aula serão introduzidas algumas das noções fundamentais no desenho de qualquer linguagem de programação. Trata-se da definição e âmbito de utilização de identificadores. A utilização de identificadores que denotam valores da linguagem é um mecanismo fundamental para a criação de abstracções numa linguagem de programação, e.g. variáveis, funções, objetos, classes, etc.

Um identificador usado numa expressão de uma linguagem denota um valor da linguagem, cuja declaração e definição é feita separadamente. A semântica das linguagens deve ser definida de tal modo que possa ser determinada a denotação de uma expressão para qualquer valor que seja o denotado pelos identificadores nela contidos. Um princípio fundamental do desenho de linguagens de programação é que o significado das expressões contendo identificadores deve ser o mesmo da expressão em que os identificadores são substituídos pelas subexpressões que eles representam (ou pelos seus valores), é o chamado **princípio da substitutividade**.

Nestas notas falamos da diferença entre literiais e identificadores, da declaração e uso de identificadores, da noção de âmbito de uma declaração, caracterizamos as ocorrências de um identificador, a noção de expressão aberta e fechada. Para além disso, introduzimos uma nova linguagem que estende a linguagem CALC (já nossa conhecida) com uma construção fundamental de declaração de identificadores.

1 Literais e identificadores

O conjunto dos nós terminais de qualquer linguagem de programação inclui necessariamente duas categorias sintáticas importantes, os literais (e.g. `true`, `false`, `[]`, `1`, `1.0`, `0xFF`) e os identificadores (e.g. x, *System.out*, *printf*).

Os literais denotam um valor que é sempre o mesmo em qualquer contexto de avaliação, por exemplo, o literal `true` ou o literal `NULL` num programa em Java terão sempre esse valor independentemente do contexto onde são

usados. Por outro lado, os identificadores denotam valores da linguagem dependendo do contexto (ou ambiente) onde são usados. Por exemplo, num comando de um programa C

```
printf("Hello, %s", name);
```

o seu efeito depende não só do valor denotado pelo identificador `name` mas também do código que está associado ao nome `printf`. Por outro lado o literal `"Hello, %s"` denota sempre a mesma cadeia de caracteres independentemente do contexto de utilização.

Para que numa dada linguagem de programação seja possível determinar a denotação de expressões com identificadores, é necessário que todos os identificadores que contribuem para essa denotação sejam associados a uma denotação própria. A essa associação chama-se ligação (*binding*). A ligação é em geral composta por uma declaração, uma definição (da sua denotação), e um contexto sintático onde a declaração é válida e onde o identificador pode ocorrer zero ou mais vezes. A esse contexto sintático chama-se âmbito (*scope*).

Como princípio universal do desenho de linguagens de programação deve-se tomar a ligação de um identificador como fixa durante toda a execução de um programa.

2 Âmbito

O âmbito de um identificador (*scope*) é o contexto sintático (zona da expressão ou programa) onde a ligação entre a declaração e a utilização de um identificador é válida, e consequentemente onde é possível determinar a denotação do mesmo. Na definição de uma função na linguagem C (C99) na Figura 1 são declarados vários identificadores

- o identificador `f` que denota a função que se está a definir. O seu âmbito são todas as definições que se seguem no mesmo ficheiro de código, e o corpo da própria função.

- o identificador `x`, que denota o parâmetro da função, uma variável local que contém o valor do argumento passado aquando da chamada da função. O âmbito da sua declaração é o corpo da função. Neste caso, excetuando o âmbito do identificador com o mesmo nome declarado dentro do ciclo.

- o identificador `z`, que denota uma variável local. O seu âmbito são todos os comandos que se seguem à sua declaração no corpo da função. Note-se que a expressão que inicializa a variável, que aparece à frente da declaração, não está incluída nesse âmbito.

```
int f(int x)
{
  int z = x+1;
  for(int j=0; j<10; j++){
    int x=j+y;
    z += x;
  }
  return z;
}
```

Figura 1: Âmbito de um identificador

```
let f x = let y = x-1 in y*x
```

Figura 2: Âmbito de um identificador (OCaml)

- o identificador `j` que denota a variável de controlo do ciclo. O seu âmbito é constituído pela condição de controlo, a expressão de incremento, e pelo corpo do ciclo (em C99). Não é visível fora do ciclo.
- o identificador `x`, denota uma variável local declarada dentro do ciclo. O seu âmbito é o restante corpo do ciclo, neste caso a afetação na linha seguinte. Não é visível fora do ciclo e esconde o parâmetro no seu âmbito.

No caso de linguagens como o C ou Java, a declaração de identificadores é apresentada juntamente com a definição de uma variável. O princípio universal de que a ligação é fixa, que a denotação de um identificador não muda durante o seu tempo de vida, mantém-se, sendo que a denotação associada ao identificador é a posição de memória correspondente e não o valor que esta contém.

Numa linguagem como a linguagem OCaml, Figura 2 a declaração de um identificador é feita pela expressão `let x = E in E'` onde o âmbito de `x` é exactamente a expressão `E'`. No caso concreto, temos uma declaração de uma função no nível principal do interpretador e:

- o identificador `f` tem por âmbito todas as expressões que forem interpretadas posteriormente no *top-level*.
- o identificador `x` que corresponde a um parâmetro, tem por âmbito o corpo da função.
- o identificador `y` tem por âmbito a expressão `y*x`.

Este é o tipo de declaração mais explícita, simples e primitiva. Note-se que a declaração `let` em OCaml não é recursiva e assim não pode ser usada na

sua própria definição. Para isso existe outra expressão de declaração que é explicitamente recursiva (`let rec`).

3 Linguagem CALCI

Assim, a linguagem exemplo que desenvolvemos será estendida com novas construções que suportam a declaração e uso de identificadores. Neste caso introduzimos a construção que declara e define um identificador e ao mesmo tempo identifica o âmbito dessa declaração, com a sintaxe concreta:

$$\texttt{decl}\ x = E\ \texttt{in}\ E'$$

onde o identificador x é associado à denotação dada pela expressão E e o âmbito onde essa ligação é a expressão E'. A expressão que representa o uso do identificador é o próprio identificador x. Os construtores indutivos da linguagem *CALCI* são:

1. `num` : $\mathit{Integer} \rightarrow \mathit{CALCI}$
2. `add` : $\mathit{CALCI} \times \mathit{CALCI} \rightarrow \mathit{CALCI}$
3. `sub` : $\mathit{CALCI} \times \mathit{CALCI} \rightarrow \mathit{CALCI}$
4. `mul` : $\mathit{CALCI} \times \mathit{CALCI} \rightarrow \mathit{CALCI}$
5. `div` : $\mathit{CALCI} \times \mathit{CALCI} \rightarrow \mathit{CALCI}$
6. `id` : $\mathit{String} \rightarrow \mathit{CALCI}$
7. `decl` : $\mathit{String} \times \mathit{CALCI} \times \mathit{CALCI} \rightarrow \mathit{CALCI}$

O tipo de dados Haskell que representa os programas da linguagem *CALCI* é descrito na que forem interpretadas posteriormente no *top-level*Figura 3

De modo a definirmos a semântica da linguagem indutivamente deveremos ter em conta que para dar um significado a uma expressão teremos que saber a denotação de todos as ocorrências de todos os identificadores.

3.1 Expressões abertas e fechadas

Um identificador pode ocorrer numa expressão de várias formas. À ocorrência de um identificador na sua declaração chama-se *ocorrência ligante* (fonte da ligação). Às várias ocorrências de um identificador no seu âmbito, chamam-se ocorrências ligadas. Uma ocorrência ligada corresponde exatamente a uma e só uma ocorrência ligante. As ocorrências de identificadores numa expressão que não correspondem a uma ocorrência ligante, chamam-se livres.

```
data CALCI =
    Num Int
  | Add CALCI CALCI
  | Sub CALCI CALCI
  | Mul CALCI CALCI
  | Div CALCI CALCI

  | Id String
  | Decl String CALCI CALCI
  deriving Show
```

Figura 3: Tipo de dados *CALCI*

```
1+2*3

3+let x = 1 in x+3

int sqr(int x) {
  return x*x;
}
```

Figura 4: Exemplos de expressões fechadas

Se existem ocorrências livres numa expressão então diz-se que essa expressão é aberta. Caso contrário diz-se que a expressão é fechada. Como exemplos podemos considerar os fragmentos de programa fechados da Figura 4, duas expressões e uma definição de função. Nestes casos é possível determinar o valor de cada um deles sem necessitar de mais informação do contexto. No caso dos fragmentos abertos da Figura 5, a sua avaliação depende das denotações de vários identificadores, nomeadamente:

- `x+f(2)` depende do valor do identificador `x` e da denotação dada ao identificador `f`, que deve ser neste caso a definição de uma função que aceita um parâmetro inteiro.
- a expressão `y + let x = 2*y in x+3` depende das denotações do identificador `y`.
- a definição da função `do_it` depende da denotação do identificador `TEN` e da denotação (código) associado ao identificador `printf`.

Assim, uma expressão diz-se fechada se o conjunto de ocorrências livres de indentificadores é vazio, ou aberta se não for. E a definição do conjunto de ocorrências livres pode ser definido indutivamente na linguagem *CALCI* da seguinte maneira:

```
x + f(2)

y + let x = 2*y in x+3

int do_it(int x) {
  for(int i = 0; i<TEN; i++)
    printf("%d\n",i);
}
```

Figura 5: Exemplos de expressões abertas

Definition 3.1 (Identificadores livres)**.** *A função free*(E) *é definida indutivamente pelos casos:*

$$
\begin{aligned}
&\mathit{free}(\mathtt{num}(n)) = \{\}\\
&\mathit{free}(\mathtt{add}(E, E')) = \mathit{free}(E) \cup \mathit{free}(E')\\
&\mathit{free}(\mathtt{sub}(E, E')) = \mathit{free}(E) \cup \mathit{free}(E')\\
&\mathit{free}(\mathtt{mul}(E, E')) = \mathit{free}(E) \cup \mathit{free}(E')\\
&\mathit{free}(\mathtt{div}(E, E')) = \mathit{free}(E) \cup \mathit{free}(E')\\
&\mathit{free}(\mathtt{id}(x)) = \{x\}\\
&\mathit{free}(\mathtt{decl}(x, E, E')) = \mathit{free}(E) \cup (\mathit{free}(E') \backslash \{x\})
\end{aligned}
$$

Com esta definição temos que $\mathit{free}(\texttt{decl}\ \ x = y + 1\ \ \texttt{in}\ \ x + z) = \{y, z\}$ e que $\mathit{free}(\texttt{decl}\ \ x = 1\ \ \texttt{in}\ \ x + 2) = \{\}$.

3.2 Semântica Operacional CALCI

A definição da semântica operacional para a linguagem *CALCI* pode ser definida utilizando a mesma técnica das aulas passadas (linguagem *CALC*), para expressões fechadas. O princípio da substitutividade deve ser tomado como base, ou seja, a denotação de uma expressão com identificadores livres é a mesma da expressão onde o identificador é substituído pela sua denotação.

No caso da linguagem *CALCI*, a denotação das expressões de declaração de identificadores, do tipo $\texttt{decl}\ \ x = E\ \ \texttt{in}\ \ E'$, é a mesma da expressão $E'\{E/x\}$, que é a expressão E' onde todas as ocorrências livres do identificador x foram substituídas pela expressão E. Note-se que no caso da expressão $\texttt{decl}\ \ x = E\ \ \texttt{in}\ \ E'$ ser fechada, de acordo com a definição da função *free*, o único identificador livre que a expressão E' pode ter é x, e a expressão E é fechada. Logo a expressão $E'\{E/x\}$ tem obrigatoriamente de ser fechada. Uma primeira tentativa de definir a semântica operacional é definida pelo código Haskell da Figura 6. Note-se que a semântica das expressões da linguagem *CALC* permanece inalterada e que a semântica da declaração é a substituição dos usos pela denotação da expressão e que o caso do identificador corresponde a um erro de execução. Observe-se que a semântica está definida para expressões fechadas e que a expressão `Id x` é

aberta. Esta semântica depende da definição de uma função de substituição de identificadores por expressões.

Definition 3.2 (Substituição)**.** *A função* $subst(E, x, E')$ *é definida indutivamente pelos casos:*

$subst(E, x, \texttt{num}(n)) = \texttt{num}(n)$
$subst(E, x, \texttt{add}(E', E'')) = \texttt{add}(subst(E, x, E'), subst(E, x, E''))$
$subst(E, x, \texttt{sub}(E', E'')) = \texttt{sub}(subst(E, x, E'), subst(E, x, E''))$
$subst(E, x, \texttt{mul}(E', E'')) = \texttt{mul}(subst(E, x, E'), subst(E, x, E''))$
$subst(E, x, \texttt{div}(E', E'')) = \texttt{div}(subst(E, x, E'), subst(E, x, E''))$
$subst(E, x, \texttt{id}(y)) = E \ \ (with\ x = y)$
$subst(E, x, \texttt{id}(y)) = \texttt{id}(y) \ \ (with\ x \neq y)$
$subst(E, x, \texttt{decl}(y, E', E'')) = \texttt{decl}(y, subst(E, x, E'), E'') \ \ (with\ x = y)$
$subst(E, x, \texttt{decl}(y, E', E'')) = \texttt{decl}(y, subst(E, x, E'), subst(E, x, E'')) \ \ (with\ x \neq y)$

Consideremos então a função de substituição, e analisemos o exemplo seguinte: `decl` $y = 3$ `in` $x + y$ e a substituição do identificador x pela expressão $y + 1$, onde a denotação do identificador y é 0. O resultado da substituição será a expressão `decl` $y = 3$ `in` $(y+1)+y$ com a denotação 7, e intuitivamente dir-se-ia que a denotação da expressão deveria ser 4. Neste caso, a diferença vem do facto da ligação da ocorrência ligada do identificador y, na expressão $y+1$, foi substituída ou capturada, por outra ocorrência ligante (com a denotação 3 em vez de 0).

Para evitar a captura de identificadores, pode fazer-se a renomeação dos identificadores declarados de modo a não colidir com qualquer outro identificador. Note-se que a expressão `decl` $x = E$ `in` E' é equivalente à expressão `decl` $x' = E$ `in` $E'\{x'/x\}$.

Assim a avaliação das expressões é melhor expressa pelo código da Figura 7. Note-se que no caso da declaração o identificador declarado é substituído por um identificador "fresco". Neste código, a produção de um identificador novo é garantido pela função `newId` baseada no monad `StateMaybe` (ver código disponibilizado), mas seria em Java ou em OCaml assegurado por um simples contador. Embora este problema de captura da ligação de identificadores possa parecer puramente académico, tal não é verdade. Este problema ocorre frequentemente em ambientes de avaliação dinâmica e de carregamento de código. Um bom exemplo é o carregamento sucessivo de ficheiros javascript numa página HTML, onde pode haver coincidências de nomes e consequentemente comportamentos imprevistos dos programas.

Como comentário final, note-se que esta semântica operacional pressupõe a rescrita das expressões (da AST), o que torna a sua implementação impraticável (ineficiente) em muitos cenários. Uma maneira de tornar essa implementação praticável e eficiente é de manter essa substituição explicitamente numa estrutura de dados à parte do código. Esse é o tema da aula que se segue.

```
eval :: CALCI -> StateMaybe Int Int

eval (Num n) = return n

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

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

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

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

eval (Decl x e e') =
     eval e''
     where e'' = subst e x e'

eval (Id x) = raise_error
```

Figura 6: Semântica operacional *CALCI* (capturing)

```
eval' :: CALCI -> StateMaybe Int Int

eval' (Num n) = return n

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

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

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

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

eval' (Decl x e e') =
     do
     x' <- newId
     e'' <- return (subst (Id x') x e')
     eval' (subst e x' e'')

eval' (Id x) = raise_error
```

Figura 7: Semântica operacional $CALCI$ (capture avoiding)

A Código Haskell

```
module CalcI where

{- ©2013 João Costa Seco, ICL DI-FCT-UNL -}

import StateMaybeMonad

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

data CALCI =
    Num Int
  | Add CALCI CALCI
  | Sub CALCI CALCI
  | Mul CALCI CALCI
  | Div CALCI CALCI

  | Id String
  | Decl String CALCI CALCI
  deriving Show

{- Examples -}

a = Id "x"

b = Decl "x" (Num 1) (Add a (Num 1))

c = Decl "x" (Id "x") b

-- decl x = 1 in x
-- decl x = x in decl x = 1 in x

{- Free identifiers -}

free :: CALCI -> [String]
free (Id x)        = [x]
free (Decl x e e') = (free e) ++ (filter (\y -> x /= y) (free e'))
free (Num n)       = []
free (Add e e')    = (free e) ++ (free e')
free (Sub e e')    = (free e) ++ (free e')
free (Mul e e')    = (free e) ++ (free e')
free (Div e e')    = (free e) ++ (free e')
```

```
{- Substitution function, does not avoid identifier capturing -}

subst :: CALCI -> String -> CALCI -> CALCI

subst e x (Id y) | x == y            = e
subst e x (Id y) | x /= y            = Id y

subst e x (Decl y e' e'') | x == y   = (Decl y (subst e x e') e'')
subst e x (Decl y e' e'') | x /= y   = (Decl y (subst e x e') (subst e x e''))

subst e x (Num n)                    = Num n
subst e x (Add e' e'')               = Add (subst e x e') (subst e x e'')
subst e x (Sub e' e'')               = Sub (subst e x e') (subst e x e'')
subst e x (Mul e' e'')               = Mul (subst e x e') (subst e x e'')
subst e x (Div e' e'')               = Div (subst e x e') (subst e x e'')

 ------------------------------------------------------------------------------
{- Semantics for closed expressions                                           -}
 ------------------------------------------------------------------------------

eval :: CALCI -> StateMaybe Int Int

eval (Num n) = return n

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

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

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

eval (Div e e') =
     do
```

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

eval (Decl x e e') = eval e''
     where e'' = subst e x e'
-- This version does not avoid capturing

eval (Id x) = raise_error


-- subst (Num 1) "x" (Id "x") == (Num 1)
-- subst (Num 1) "y" (Id "x") == (Id "x")

 ------------------------------------------------------------------------------
{- Semantics for closed expressions and capture avoiding substitutions         -}
 ------------------------------------------------------------------------------

eval' :: CALCI -> StateMaybe Int Int

eval' (Num n) = return n

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

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

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

eval' (Div e e') =
     do
     l <- (eval' e)
```

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

eval' (Decl x e e') =
     do
     x' <- newId
     e'' <- return (subst (Id x') x e')
-- replacing the declared identifier by a new one to avoid capturing of identifiers
     eval' (subst e x' e'')

eval' (Id x) = raise_error

runstate :: CALCI -> Maybe Int
runstate e = let SM(f) = (eval' e) in snd (f 0)
```