# Interpretation and Compilation of Programming Languages
# Part 4 - Name declaration and binding

João Costa Seco

March 26, 2014

In this lecture we discuss a fundamental component of programming languages, which is the notion of variable. The notion of variable (from a mathematical point of view) allows for the expression of computations and properties by abstracting the actual components. Note that the notion of variable, in this context, is different from the notion of state variable, which is linked to the use of memory in imperative languages.

Variables as introduced in this lecture are a generic mechanism, important in all languages to define (name) language abstractions like state variables, functions, classes, etc..

We introduce the fundamental notions of binding, scope, and environment, occurrences of identifiers, open expressions, and closed expressions.

We introduce the fundamental declaration operation and the notion of scope of a variable and evaluation environment. Also, we define the semantics of a small language with declarations, in three different steps. This is a way of studying the difference between dynamic and static binding of variables.

In the end we define a type semantics for the language with declarations that ensures that no runtime errors occur related to undefined variables.

## 1 Literals and variables

The semantics of a programming language with variables must be defined in such a way that the denotation for a variable is determined according to its declaration. We follow the fundamental principle of substitution, that states

> *the semantics of an expression, containing declarations of variables (or identifiers), should yield the same result that the expres-*

*sion obtained by replacing all variables by the expressions used to define them (or their meanings).*

Consider the non-terminals of a programming language, many denote some a value, are literals or variables. From now on we call identifiers to variables to avoid any possible misunderstandings when state variables are introduced.

On the one hand, literals always denote the same value, independent from the evaluation context, (e.g. `true`, `false`, `[]`, `1`, `1.0`, `0xFF`, `NULL`) and, on the other hand, the denotation of identifiers (e.g. $x$, *System.out*, *printf*) depend on the evaluation context. For instance in the programming language C,

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

The effect of the statement above depends, not only on the meaning of identifier `name`, but also on the denotation of the identifier `printf`. On the other hand, the string literal `"Hello, %s"` denotes the same array of characters independently of the evaluation context.

# 2 Binding and scope

In order to define the semantics of a language with identifiers we first need to understand the notion of *binding*. A *binding* is established between the declaration, and a set of occurrences of an identifier. In order to follow the substitution principle, the binding of an identifier must be fixed in all executions of a particular program.

Along with the declaration, each language establishes the fragment of code where each declared identifier is visible, i.e. where all occurrences of a declared identifier are bound to its declaration. This is known as the scope of an identifier.

The scope of an identifier (of its declaration) is the syntactically context where an occurrence of the identifier is bound to it. As a consequence, the scope of an identifier is the fragment where the corresponding denotation can be found deterministically. For instance, the function definition in `C` language (`C99`) of Figure 1, there are several occurrences of identifiers:

- of identifier `f`, that denotes the function being defined. The scope of identifier `f` is the remaining file and the body of the current definition.
- of identifier `x`, that denotes a function parameter, that is considered as a local variable initialised with the argument value. The scope of the parameter is the body of the function. In this case the scope excludes the scope of identifier `x` declared inside the loop.

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

Figure 1: Scope (`C`).

- of identifier `z`, that denotes a local variable. Its scope are all statements following its declaration, inside the function body. Note that the initialiser expression is not included in the scope.
- of identifier `j`, that denotes the loop's control variable. Its scope is the condition of the loop, the increment expression, and the body of the loop (in `C99`). It is not visible outside the loop.
- of identifier `x`, that denotes a local variable inside the loop. Its scope are the statements that follow its declaration, inside the loop. In this case, it is the assignment in the next line. It is not visible outside the loop, and its scope shadows the function parameter `x`.

In many imperative languages, like `C` or `Java`, the declaration of identifiers is linked with the definition of state variables. The universal principal of static binding, that states that the denotation of an identifier does not change during its lifetime is still true. In this case, the denotation of the identifier that we refer to is the given memory location. Although its contents may change, the actual memory address is maintained.

Most languages also provide mechanisms that detach the declaration of identifiers from the definition of its denotations. Function prototypes in `C`, class prototypes in `Java`, module interfaces in `OCaml`, etc.

In functional languages like `OCaml`, the declaration of an identifier is, in general, linked to its definition in a `let` expression, of the form `let x = E in E'`. In this case, the scope of identifier `x` is expression `E'`.

Consider the code in Figure 2 containing a function declaration at the interpreter top level where:

- the scope of identifier `f` contains all expressions in the remainder of the file, or given in the interpreter's *top-level*.

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

Figure 2: Scope (OCaml)

- identifier `x` is a parameter and its scope is the function body.
- identifier `y` is visible in expression `y*x`.

This kind of declaration is explicit and primitive, and taking advantage of the fact that functions are first-class values in OCaml. In other languages declaration is syntactically connected to definitions (e.g. functions in C).

Notice that a `let` declaration in `OCaml` is not recursive. For that purpose, in the `OCaml` language there is an explicit construct (`let rec`).

# 3 Language CALCI

We now define a new language, that extends language `CALC`, and supports the primitive identifier declaration and use of identifiers. We introduce a declaration of identifiers with the concrete syntax:

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

where identifier $x$ is associated to the denotation of expression $E$ and its scope is expression $E'$. The identifier expression has the given denotation in its scope. The constructions of language *CALCI* are defined by the abstract data type:

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` : $\mathit{String} \times \mathit{CALCI} \times \mathit{CALCI} \rightarrow \mathit{CALCI}$

The `Haskell` data type, depicted in Figure 3, that represents all *CALCI* programs.

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

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

Figure 3: Tipo de dados *CALCI*

## 3.1 Open and closed expressions

It is only possible to assign a denotation to an arbitrary expression if we know exactly what is the denotation of each identifier in it.

The occurrences of an identifier in an expression can be classified in three different ways:

- A **binding occurrence** is an occurrence at the declaration point. It is the source of all bindings for that identifier declaration.
- A **bound occurrence** is an occurrence of an identifier within the scope of a declaration. Each bound occurrence is linked to one particular declaration.
- A **free occurrence** is an occurrence that is not linked to any declaration.

An expression with no free occurrences is called a **closed expression**, otherwise it is called an **open expression**. Take the examples of the closed code fragments in Figure 4. In these cases, we can deterministically obtain a denotation without any further context information. In the cases of open expressions, in Figure 5, we cannot determine a denotation. The denotation of such expressions depends on the denotation of its free identifiers, as follows:

- `x+f(2)` depends on the value of identifier `x` and identifier `f`. In this case `f` should denote a function that takes an integer value as argument.
- expression `y + let x = 2*y in x+3` depends on the denotation of identifier `y`.

```
1+2*3

3+let x = 1 in x+3

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

Figure 4: Closed expressions

```
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);
}
```

Figure 5: Open expressions

- The definition of function `do_it` depends on the denotation of identifier `TEN` and the denotation of identifier `printf` (it's code).

In order to formally define the notion of free and closed expressions, we compute the set of free identifiers of an expression. A closed expression has an empty set of free identifiers, it is otherwise open.

The inductive definition for the set of free identifiers for language *CALCI* is the following:

**Definition 3.1** (Free identifiers)**.** *The set of free identifiers, written free*$(E)$*, is inductively defined by the cases:*

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

We have that $\mathit{free}(\texttt{decl}\ \ x = y{+}1\ \ \texttt{in}\ \ x{+}z) = \{y, z\}$ and that $\mathit{free}(\texttt{decl}\ \ x = 1\ \ \texttt{in}\ \ x + 2) = \{\}$.

## 3.2 Operational Semantics for CALCI

The operational semantics for language *CALCI* is defined for closed expressions as in previous lectures.

Taking the substitution principle as guide we have that the denotation of an expression is obtained by replacing the occurrences of an identifier by its denotation.

In the case of expressions of the form `decl` $x = E$ `in` $E'$, the denotation is the same as expression $E'\{E/x\}$, that is obtained by replacing all free occurrences of identifier $x$ in $E'$ by expression $E$. Note that if expression `decl` $x = E$ `in` $E'$ is closed (and we assume that it is), the only free identifier that can occur in expression $E'$, is $x$. So, expression $E'\{E/x\}$ must be closed.

We define the operational semantics through the Haskell code in Figure 6.

Notice that the semantics of expressions of language *CALC* remains untouched, and that the semantics of the declaration is defined by the substitution of the identifier by the expression. Note also that the denotation of a (free) identifier is an error. The semantics is only defined for closed expressions and that expression `Id x` is open. The semantics depends on the substitution function of identifiers by expressions.

**Definition 3.2** (Substitution)**.** *Function* $subst(E, x, E')$ *is inductively defined by the cases:*

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

Consider the substitution function and the following example and the following example: `decl` $y = 3$ `in` $x + y$. The substitution of identifier $x$ by expression $y + 1$, with denotation of $y$ being 0. The result of the substitution is expression `decl` $y = 3$ `in` $(y + 1) + y$ with denotation 7. Intuitively, we expected the denotation 4 for the expression. The discrepancy comes from the fact that the binding of identifier $y$, in expression $y + 1$, was captured by another declaration with denotation 3 instead of 0.

To avoid the capturing of identifiers we need to rename the declared identifiers. Note that expression `decl` $x = E$ `in` $E'$ is equivalent to `decl` $x' = E$ `in` $E'\{x'/x\}$.

The code in Figure 7 implements this semantics. Note that in the case of the declaration, the declared identifier is replaced by a *fresh* identifier.

In this piece of code, a new identifier is ensured by function `newId` based on monad `StateMaybe` (see appendix), It should be ensured by a simple counter in `Java` or `C`. This substitution with capturing of identifiers is more common than we might think at first. It occurs often in cases like sequential loading of Javascript files in an HTML file, where coincidences of names can occur.

```
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
```

Figure 6: operational semantics of *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
```

Figure 7: operational semantics $CALCI$ (capture avoiding)