

***Programming Language Concepts and Semantics
Part II***

***Lecture XII
Expressions and Declarations***

In this lecture we will look at fundamental concepts of expressions (constants, operations, identifiers, etc) and declarations (bindings and their scopes).

The lecture follows Chapters 5 and 6 of “Fundamental Concepts and Formal Semantics of Programming Languages, Lecture Notes by Peter Mosses. University of Aarhus”

Expressions: Fundamental Concepts

An *expression* E is a construct which normally computes a value

$$E : \text{Exp}$$

as opposed to commands (which do not compute values) and declarations (which compute bindings).

Example: in ML commands are expressions which compute the null value of the unit type.

Values might be simple or structured (formed by sub-components).

The evaluation of an expression:

- need not involve evaluation of all its subexpressions
- might terminate abnormally or not at all
- might have side effects
- in ML is sequential (left-to-right, top-to-bottom)

Expressions: Fundamental Concepts (Cont.)

A *constant* is an expression whose value does not change. Usually they include numbers, characters, strings, and booleans.

We will assume constants are replaced by their values in the mapping from concrete to abstract syntax, and include the corresponding set of values directly in Exp , (thus $\text{Exp} ::= \text{Integer} \mid \text{Boolean} \mid \dots$).

Example: in ML $()$ and $[]$ are constants; strings and characters are constants; Booleans are identifiers with fixed bindings.

An *operation* $O:Op$ has a fixed interpretation as a function applied to the value of its argument expressions (this value must be computed first). In higher order languages (such as ML), operations can themselves be used as expressions ($\text{Exp} ::= \text{app}(Op, Arg)$).

Multiargument ops will be considered as unary ops on tuples. Predicates are regarded as boolean-valued ops ($Op ::= =$)

Operations on given sets can be included as abstract constructs.

Example: $Op ::= + \mid - \mid * \mid \text{div} \mid \text{mod} \mid < \mid = < \mid > \mid \dots$

An *identifier* $I:Id$ is bound to a value through a declaration. Values of identifiers at a particular point in a computation are called *current bindings*.

Identifiers may already be bound. **Example:** symbols `true`, `=` and `+` which are constants or operations in some languages, are identifiers in ML. We thus need to be able to apply ids to arguments ($\text{Exp} ::= \text{app}(Id, Arg)$).

Expressions: Fundamental Concepts (Cont.)

A *tuple* is an expression which collects the value of its components (thus $\text{Exp} ::= \text{tup}(\text{Exp}^*)$).

Sequences and tuples are similar but sequences cannot nest. Here tup forms a tuple from the sequence Exp^* .

Evaluation of each sub-expression may be sequential ($\text{Exp} ::= \text{tup-seq}(\text{Exp}^*)$) with subexpressions evaluated left-to-right.

In ML:

- components can be selected from a tuple (as a record) using:

$$\text{Op} ::= \text{nth}(\text{Integer})$$

- lists can be represented as flat trees. If list is an operation, then $[\text{E}_1, \dots, \text{E}_n]$ is treated as:

$$\text{app}(\text{list}, \text{tup-seq}(\text{E}_1, \dots, \text{E}_n))$$

The empty list is the value of the `nil` identifier

The data constructor `::` is interpreted in MSOS as the operation `cons`.

Conditional expressions are special cases of the generic conditional construct:

$$\text{Exp} ::= \text{cond}(\text{Exp}, \text{Exp}, \text{Exp})$$

Note that this conditional construct could be considered itself a special case of *case selection*

Expressions: Formal Semantics

Let us now introduce the MSOS modules that define the dynamic semantics of the previous abstract constructs.

Module *Cons/Exp*:

State ::= Exp

Final ::= Value

Exp ::= Value

Recall that expressions get replaced by their computed values. Each set of constant values has to be included in `Value`. For example:

Module *Cons/Exp/Integer*:

Value ::= Integer

which ensures it is also included in `Final`.

Expressions: Formal Semantics (Cont.)

The semantics of ops is specified in terms of their application:

Module *Cons/Exp/app-Op*:

Passable ::= Value

Value ::= tup(Value*)

$$\text{ARG} \text{ --}\{\dots\}\text{--> ARG'}$$

app(O, ARG) : EXP --{...}--> app(O, ARG')

$$O(V^*) = V'$$

app(O, tup(V*)) : EXP ----> V'

$$O(V) = V'$$

app(O, V) : EXP ----> V'

Expressions: Formal Semantics (Cont.)

The semantics of identifiers requires labels with the current environment:

Module *Cons/Id*:

State ::= Id

Final ::= Bindable

Id ::= Bindable

Label = {env:Env, ...}

lookup(I, ENV) = BV

I:Id --{env=ENV, ---}> BV

not def lookup(I, ENV),
init-end = ENV',
lookup(I, ENV') = BV

I:Id --{env=ENV, ---}> BV

Recall that ENV:Env = (Id, Bindable)Map.

The operation lookup(I, ENV) **is undefined when** ENV **has no binding for** I

The environment init-env **is left open. If empty, the second rule is unapplicable.**

Expressions: Formal Semantics (Cont.)

Evaluation of an identifier in an expression involves obtaining its bound value:

Module *Cons/Exp/Id*:

$I \text{ ---} \rightarrow BV$

 $I : \text{Exp} \text{ ---} \rightarrow BV$

For the case in which the identifier is an operation:

Module *Cons/Exp/app-Id*:

$I \text{ ---} \rightarrow I'$

 $\text{app}(I, \text{ARG}) : \text{Exp} \text{ ---} \rightarrow \text{app}(I', \text{ARG})$

$\text{ARG} \text{ --}\{\dots\}\text{-} \rightarrow \text{ARG}'$

 $\text{app}(I, \text{ARG}) : \text{Exp} \text{ --}\{\dots\}\text{-} \rightarrow \text{app}(I, \text{ARG}')$

Expressions: Formal Semantics (Cont.)

Components of a tuple may be evaluated in any order:

Module *Cons/Exp/tup*:

Value ::= tup(Value*)

$$\begin{aligned} E+ &= (E1^*, E2+), E2+= (E, E3^*), \\ &E \text{ --}\{\dots\}\text{--} E' \\ (E', E3^*) &= E2'+, (E1^*, E2'+) = E', \end{aligned}$$

tup(E+) : Exp --{\dots}--> tup(E'+)

by extracting any component E from the sequence $E+$, making a transition, and reconstructing the sequence $E'+$

Sequential tuples require an extra step:

Module *Cons/Exp/tup-seq*:

Value ::= tup(Value*)

$$\begin{aligned} E+ &= (V1^*, E2+), E2+= (E, E3^*), \\ &E \text{ --}\{\dots\}\text{--} E' \\ (E', E3^*) &= E2'+, (V1^*, E2'+) = E', \end{aligned}$$

tup-seq(E+) : Exp --{\dots}--> tup-seq(E'+)

tup-seq(V*) : Exp ---> tup(V*)

$V1^*$ ensures all previous expressions have been evaluated.

Expressions: Formal Semantics (Cont.)

The semantics of conditional constructs is very simple:

Module *Cons/Exp/cond*:

Value ::= Boolean

$$E \text{ --}\{\dots\}\text{--> } E'$$

cond(E,E1,E2):Exp --{...}-> cond(E',E1,E2)

cond(true,E1,E2):Exp ----> E1

cond(false,E1,E2):Exp ----> E2

Declarations: Fundamental Concepts

A *declaration* $D:Dec$ is a construct which computes bindings.

A *binding* is an association between an identifier and a bindable value. In ML all expressions are bindable, but that is rare.

Its *scope* is the program part where the binding is current, usually determined by the context-free structure of the program. Holes in the scope appear when the binding is overridden by another binding for the same id.

A *binding* occurrence is used to *create* a binding. A *bound/free* occurrence is used to *refer* to a binding present/not present in the construct.

A part of the program without free occurrences is said to be *closed*.

The same identifier might refer to more than one binding, if the context of the binding determines which of the bindings will be selected.

Declarations: Fundamental Concepts (Cont.)

A *value* declaration binds an identifier to the value of an expression:

$$\text{Dec} ::= \text{bind}(\text{Id}, \text{Exp})$$

A *local* declaration might override non-local bindings.

$$\text{Exp} ::= \text{local}(\text{Dec}, \text{Exp})$$

The expression $\text{local}(D, E)$ is written `let D in E end` in ML, and restricts the bindings computed by D to the evaluation of expression E (which can only start once all bindings in D have been computed).

$$\text{Dec} ::= \text{local}(\text{Dec}, \text{Dec})$$

The declaration $\text{local}(D1, D2)$ is written `local D1 in D2 end` in ML and is used for defining functions in $D1$ which are local (auxiliary) to those in $D2$.

An *accumulating* declaration may override previous bindings, and has unlimited scope:

$$\text{Dec} ::= \text{accum}(\text{Dec}, \text{Dec})$$

The declaration $\text{accum}(D1, D2)$ is written `D1;D2` in ML and is very similar to $\text{local}(D1, D2)$, but while local ends up discarding the bindings computed by $D1$, $\text{accum}(D1, D2)$ combines both ($D2$ overrides $D1$ in case of overlap).

Declarations: Fundamental Concepts (Cont.)

A *simultaneous* declaration computes bindings independently

`Dec ::= simult(Dec,Dec)`

i.e., `simult(D1,D2)` excludes D2 from the scope of the bindings computed by D1 and vice-versa. The bindings are subsequently combined by using union. Overlapping domains result in undefined behaviour.

The sequentialised variant

`Dec ::= simult-seq(Dec,Dec)`

ensures bindings D1 in `simult-seq(D1,D2)` are computed first.

Declarations: Formal Semantics

Let us now introduce the MSOS modules that define the dynamic semantics of the previous abstract constructs.

Module *Cons/Dec*:

State ::= Dec

Final ::= Env

Dec ::= Env

We will need the following operations on maps:

- $I \mid \rightarrow BV$ **maps I to BV**
- ENV / ENV' **lets ENV override ENV'**
- $ENV + ENV'$ **unions them (undefined if there is overlap)**
- $ENV - IS$ **removes from ENV bindings of any identifier in IS**

Module *Cons/Dec/bind*:

$$E \text{ --}\{\dots\}\text{--} \rightarrow E'$$

 $\text{bind}(I, E) : \text{Dec} \text{ --}\{\dots\}\text{--} \rightarrow \text{bind}(I, E')$

$\text{bind}(I, BV) : \text{Dec} \text{ ---} \rightarrow \{I \mid \rightarrow BV\}$

Note that values computed by expressions do not need to be bindable, however, the scnd rule is only applicable if the value for the expression is bindable. In ML, Values would be in Bindable (values are first class objects).

Declarations: Formal Semantics (Cont.)

The env component of a label represents the current bindings.

Module *Cons/Exp/local*:

Label = {env:Env, ...}

$D \text{ --}\{\dots\}\text{--> } D'$

 $\text{local}(D, E) : \text{Exp} \text{ --}\{\dots\}\text{--> } \text{local}(D', E)$

$(\text{ENV}/\text{ENV0}) = \text{ENV}', E \text{ --}\{\text{env}=\text{ENV}', \dots\}\text{--> } E'$

 $\text{local}(\text{ENV}, E) : \text{Exp} \text{ --}\{\text{env}=\text{ENV0} \dots\}\text{--> } \text{local}(\text{ENV}, E')$

$\text{local}(\text{ENV}, V) : \text{Exp} \text{ ---> } V$

First, all bindings in D are computed. Then, the ENV0 obtained from D overrides ENV while evaluating E . Once evaluation of E in $\text{local}(D, E)$ has finished, local bindings can be disregarded.

Declarations: Formal Semantics (Cont.)

Rules for `local(D1,D2)` are analogous. Module *Cons/Dec/local*:

Label = {env:Env, ...}

$D1 \text{ --}\{\dots\}\text{--> } D1'$

`local(D1,D2):Dec --{...}-> local(D1',D2)`

$(ENV/ENV0) = ENV', D2 \text{ --}\{\text{env}=ENV', \dots\}\text{--> } D2'$

`local(ENV,D2):Dec --{env=ENV0...}-> local(ENV,D2')`

`local(ENV1,ENV2):Dec ---> ENV2`

Accumulating declarations are very similar. Module *Cons/Dec/accum*:

Label = {env:Env, ...}

$D1 \text{ --}\{\dots\}\text{--> } D1'$

`accum(D1,D2):Dec --{...}-> accum(D1',D2)`

$(ENV/ENV0) = ENV', D2 \text{ --}\{\text{env}=ENV', \dots\}\text{--> } D2'$

`accum(ENV,D2):Dec --{env=ENV0...}-> accum(ENV,D2')`

$(ENV2/ENV1) = ENV$

`accum(ENV1,ENV2):Dec ---> ENV`

The only difference is in the last rule.

Declarations: Formal Semantics (Cont.)

Computation of simultaneous declarations may be interleaved:

Module *Cons/Dec/simult*:

Label = {env:Env, ...}

$D1 \text{ --}\{\dots\}\text{--> } D1'$

simult(D1,D2):Dec --{...}-> simult(D1',D2)

$D2 \text{ --}\{\dots\}\text{--> } D2'$

simult(D1,D2):Dec --{...}-> simult(D1,D2')

$(ENV2+ENV1) = ENV$

simult(ENV1,ENV2):Dec ----> ENV

The rules for `simult-seq(D1,D2)` differ only in having `ENV1` instead of `D1` in the second rule (to ensure `D1` is computed first).