Lecture: Big-Step Semantics

1 Representing Abstract Syntax

These are examples of *arithmetic expressions*:

- `2 * 4`
- `1 + 2 + 3`
- `5 * 4 * 2`
- `1 + 2 * 3`

We all know how to evaluate these expressions in our heads. But, when we do, we resolve several ambiguities. For example, should we evaluate `1 + 2 * 3` like this:

```
     1 + 2 * 3
  =  (1 + 2) * 3
  =  3 * 3
  =  9
```

or like this:

```
     1 + 2 * 3
  =  1 + (2 * 3)
  =  1 + 6
  =  7
```

The latter is the convention in mathematics, but it is an arbitrary choice. A programming language could adopt either convention or adopt a completely different notation. For example, the following three programs, written in three different languages, all evaluate to the same value:

- `1 + 2` infix syntax, from the C language
- `(+ 1 2)` parenthesized prefix syntax, from the Scheme language
- `1 2 +` postfix syntax, from the Forth language

These three *concrete syntaxes* are very different, but all mean "the sum of the number one and the number two".

Concrete syntax is important, because it is the human-computer interface to a programming language. It is easy to find acrimonious debates on the Web about the virtues of Python's indentation-sensitive syntax, whether semicolons should be optional in JavaScript, how C code should be indented, and so on. But, **this course will almost completely ignore concrete syntax** because it is irrelevant to the semantics of programming languages. We will instead work with *abstract syntax*, which is an abstract "tree-shaped" representation of syntax that suffers none of the ambiguities of concrete syntax. In compilers, the *parser* consumes a concrete-syntax string and produces an equivalent *abstract syntax tree.* This course will largely ignore parsing and instead work directly with the abstract syntax tree.

OCaml makes it easy to define a type that represents the abstract syntax of arithmetic expressions:

```
type exp =
  | Num of int
  | Add of exp * exp
  | Mul of exp * exp
  | Div of exp * exp
```

$$\begin{aligned} e ::=\ & \mathsf{true} \\ \mid\ & \mathsf{false} \\ \mid\ & n \\ \mid\ & e_1 + e_2 \\ \mid\ & e_1 * e_2 \\ \mid\ & e_1 > e_2 \\ \mid\ & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ v ::=\ & \mathsf{true} \\ \mid\ & \mathsf{false} \\ \mid\ & n \end{aligned}$$

(a) Syntax.

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

$$\text{True}\ \frac{}{\mathsf{true} \Downarrow \mathsf{true}} \qquad \text{False}\ \frac{}{\mathsf{false} \Downarrow \mathsf{false}}$$

$$\text{Add}\ \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n_3 = n_1 + n_2}{e_1 + e_2 \Downarrow n_3} \qquad \text{Mul}\ \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n_3 = n_1 \cdot n_2}{e_1 * e_2 \Downarrow n_3}$$

$$\text{GT-True}\ \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n_1 > n_2}{e_1 > e_2 \Downarrow \mathsf{true}} \qquad \text{GT-False}\ \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n_1 \leq n_2}{e_1 > e_2 \Downarrow \mathsf{false}}$$

$$\text{If-True}\ \frac{e_1 \Downarrow \mathsf{true} \quad e_2 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v} \qquad \text{If-False}\ \frac{e_1 \Downarrow \mathsf{false} \quad e_3 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}$$

(b) Semantics.

Figure 4.1: Syntax and semantics of a language with arithmetic and boolean expressions.

Here are some examples of abstract arithmetic expressions and their concrete representations (written in normal, mathematical notation):

Concrete Syntax	Abstract Syntax
`1 + 2 + 3`	`Add (Add (Num 1, Num 2), Num 3)`
`1 + 2 * 3`	`Add (Num 1, Mul (Num 2, Num 3))`
`(1 + 2) / 3`	`Div (Add (Num 1, Num 2), Num 3)`

2 Syntax as Sets

We start with a very rigorous mathematical definition of the abstract syntax of a small language of arithmetic and boolean expressions.

Definition 1 (Syntax of arithmetic expressions)**.** *Let $\mathcal{E}$ denote the set of arithmetic expressions. We define $\mathcal{E}$ to be the smallest set that is generated by the following rules:*

- $\mathsf{true} \in \mathcal{E}$.
- $\mathsf{false} \in \mathcal{E}$.
- *If* $n \in \mathbb{Z}$ *then* $n \in \mathcal{E}$.
- *If* $e_1 \in \mathcal{E}$ *and* $e_2 \in \mathcal{E}$ *then* $e_1 + e_2 \in \mathcal{E}$.
- *If* $e_1 \in \mathcal{E}$ *and* $e_2 \in \mathcal{E}$ *then* $e_1 * e_2 \in \mathcal{E}$.
- *If* $e_1 \in \mathcal{E}$ *and* $e_2 \in \mathcal{E}$ *then* $e_1 > e_2 \in \mathcal{E}$.
- *If* $e_1 \in \mathcal{E}$, $e_2 \in \mathcal{E}$, *and* $e_3 \in \mathcal{E}$ *then* $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \in \mathcal{E}$.

Hopefully, you'll agree that defining syntax in this way is extremely tedious. We will never do this again and instead define syntax using the notation in fig. 4.1a.

This notation is much terser and is what you'll find when you read the programming languages literature. But, you should be aware that it is just shorthand for the more verbose definition given above. Also note that we are "abusing notation" and using the metavariable e to denote the set of expressions and elements of the set (and similarly for n). This is standard practice.

$\boxed{e}$

$$\begin{array}{rl} e ::= & \cdots \\ \mid & x \\ \mid & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \end{array}$$

(a) Syntax.

$\boxed{e \Downarrow v}$

$$\text{Let}\ \frac{\begin{array}{c} e_1 \Downarrow v_1 \\ e_2[x \mapsto v_1] \Downarrow v_2 \end{array}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2}$$

(b) Semantics.

$\boxed{e[x \mapsto v] = \mathrm{e}}$

$$\begin{array}{lll}
\mathsf{true}[x \mapsto v] & = \mathsf{true} & \\
\mathsf{false}[x \mapsto v] & = \mathsf{false} & \\
n[x \mapsto v] & = n & \\
e_1 + e_2[x \mapsto v] & = e_1[x \mapsto v] + e_2[x \mapsto v] & \\
e_1 * e_2[x \mapsto v] & = e_1[x \mapsto v] * e_2[x \mapsto v] & \\
e_1 > e_2[x \mapsto v] & = e_1[x \mapsto v] > e_2[x \mapsto v] & \\
\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3[x \mapsto v] & = \mathsf{if}\ e_1[x \mapsto v]\ \mathsf{then}\ e_2[x \mapsto v]\ \mathsf{else}\ e_3[x \mapsto v] & \\
x[x \mapsto v] & = v & \\
y[x \mapsto v] & = y & \text{when } x \neq y \\
\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2[x \mapsto v] & = \mathsf{let}\ x = e_1[x \mapsto v]\ \mathsf{in}\ e_2 & \\
\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2[x \mapsto v] & = \mathsf{let}\ y = e_1[x \mapsto v]\ \mathsf{in}\ e_2[x \mapsto v] & \text{when } x \neq y
\end{array}$$

(c) Substitution.

Figure 4.2: Identifiers and let-expressions. Extends fig. 4.1.

3 Semantics as Relations

Inference rules In this section, we use *inference rules* to define the semantics of our language. These are some examples of inference rules:

$$\frac{X}{Y} \quad \frac{\begin{array}{c} A \\ B \end{array}}{C} \quad \frac{}{W}$$

You should read these rules as:

- If X holds, then Y holds.
- If A and B hold, then C holds.
- W holds (i.e., W is an axiom).

Semantics Programs evaluate expressions until they become *values*. Intuitively, a value is an expression that cannot be further simplified. For the language defined above, the values (v) are just integers and booleans. Note that $v \subset e$.

Given our definition of expressions and values, we can define the semantics of the language as an *evaluation relation*. The evaluation relation is a binary relation that relates expressions e to equivalent values v. We write the evaluation relation using the following notation:

$$e \Downarrow v$$

You should pronounce this as "e evaluates to v".

The $e \Downarrow v$ relation is defined by a set of inductively defined rules, similar to the definition of expressions. However, instead of first soldiering through a long definition in prose, we'll use the inference rules in fig. 4.1b to define $e \Downarrow v$. Notice that $e \Downarrow v$ is a partial function. E.g., $\mathsf{true} + 5$ is not defined. Moreover, in a richer language with non-deterministic features, such as threads or random number generators, $e \Downarrow v$ may be a relation.

4 Let-Binding and Substitution

Even when we write simple arithmetic expressions, we often need to reuse the same expression in several places. If we try to repeat the same expression, we are likely to propagate mistakes. More fundamentally, repeated expressions consume more storage space and require more steps to evaluate. Therefore, all programming languages provide a means to name expressions.

Figure 4.2a extends the syntax of our language with *let-expressions* and *identifiers*. A let-expression, such as $\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ \mathsf{x} * \mathsf{x}$ has three parts:

- A *binding identifier* x,
- A *named expression* $1 + 2$, and
- The *body* $x * x$.

In this example, the body has two bound identifiers that refer to the binding identifier.

The semantics of this language extension are given in fig. 4.2b. Notice that there is no inference rule for evaluating identifiers. Instead, the inference rule for let expressions *substitutes* identifiers with the value of the named expression. The substitution function, $e[x \mapsto v]$, is typically read as "substitute x with v in e".

Here is an example of a derivation that uses substitution:

$$\textsc{Let}\ \dfrac{\textsc{Add}\ \dfrac{\overset{\textsc{Num}}{1 \Downarrow 1} \qquad \overset{\textsc{Num}}{2 \Downarrow 2}}{1 + 2 \Downarrow 3} \qquad (x * x)[x \mapsto 3] = 3 * 3 \qquad \textsc{Mul}\ \dfrac{\overset{\textsc{Num}}{3 \Downarrow 3} \qquad \overset{\textsc{Num}}{3 \Downarrow 3}}{3 * 3 \Downarrow 9}}{\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ x * x \Downarrow 9}$$

This is a very straightforward example since it only has one let-expression. In general, a program may have several let-expressions and even reuse the same identifier in several places. The semantics has been carefully designed to handle these cases in a natural way.

This example defines two names by nesting let-expressions:

$$\textsc{Let}\ \dfrac{\vdots \qquad \textsc{Let}\ \dfrac{\vdots}{\mathsf{let}\ y = 20 + 10\ \mathsf{in}\ y * x \Downarrow 300}}{\mathsf{let}\ x = 10\ \mathsf{in}\ \mathsf{let}\ y = 20 + x\ \mathsf{in}\ y * x \Downarrow 300}$$

This example has two let-expressions that use the same name:

$$\textsc{Let}\ \dfrac{\overset{\textsc{Num}}{10 \Downarrow 10} \qquad (\mathsf{let}\ x = 20\ \mathsf{in}\ 1 + x)[x \mapsto 10] = \mathsf{let}\ x = 20\ \mathsf{in}\ 1 + x \qquad \textsc{Let}\ \dfrac{\vdots}{\mathsf{let}\ x = 20\ \mathsf{in}\ 1 + x \Downarrow 21}}{\mathsf{let}\ x = 10\ \mathsf{in}\ \mathsf{let}\ x = 20\ \mathsf{in}\ 1 + x \Downarrow 21}$$

As the proof tree shows, the identifier in the expression $1 + x$ is bound to the inner definition of x. We say that the inner x *shadows* the enclosing definition.

4.1 Substitution and Evaluation Order

The inference rule for let-expressions first evaluates the named expression to a value and then substitutes that value into the body. Consider this alternate definition, which applies substitution first:

$$\textsc{Let'}\ \dfrac{e_2' = e_2[x \mapsto e_1] \qquad e_2' \Downarrow v}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v}$$

For example, the following proof that $\mathsf{let}\ x = 10 + 10\ \mathsf{in}\ x * x \Downarrow 400$ uses this alternate rule for let. Notice that we use the Add rule twice and the Mul rule once:

$$\textsc{Let'}\ \dfrac{x * x[x \mapsto 10 + 10] = (10 + 10) * (10 + 10) \qquad \textsc{Mul}\ \dfrac{\textsc{Add}\ \dfrac{\textsc{Num}\ 10 \Downarrow 10 \qquad \textsc{Num}\ 10 \Downarrow 10}{10 + 10 \Downarrow 20} \qquad \textsc{Add}\ \dfrac{\textsc{Num}\ 10 \Downarrow 10 \qquad \textsc{Num}\ 10 \Downarrow 10}{10 + 10 \Downarrow 20}}{(10 + 10) * (10 + 10) \Downarrow 400}}{\mathsf{let}\ x = 10 + 10\ \mathsf{in}\ x * x \Downarrow 400}$$

However, using the original Let rule, we use the the Add and Mul rules once each:

$$\textsc{Let}\ \dfrac{\textsc{Add}\ \dfrac{\textsc{Num}\ 10 \Downarrow 10 \qquad \textsc{Num}\ 10 \Downarrow 10}{10 + 10 \Downarrow 20} \qquad x * x[x \mapsto 20] = 20 * 20 \qquad \textsc{Mul}\ \dfrac{\textsc{Num}\ 20 \Downarrow 20 \qquad \textsc{Num}\ 20 \Downarrow 20}{20 * 20 \Downarrow 400}}{\mathsf{let}\ x = 10 + 10\ \mathsf{in}\ x * x \Downarrow 400}$$

Loosely speaking, the larger proof tree corresponds to an evaluation that "takes more time". In fact, proofs with Let' will be larger whenever there are multiple occurrences of the identifier in the body of the let expression.

However, a deeper problem arises when the named expression is faulty. For example, let e be the expression $\mathsf{let}\ x = \mathsf{true} + 10\ \mathsf{in}\ 200$. Using the Let rule, there does not exist a v, such that $e \Downarrow v$. However, using the Let' rule, $e \Downarrow 200$, since x is unused in the body.

We'll investigate this in more depth in a few weeks.

```
type id = string

type exp =
  | Num of int
  | Bool of bool
  | Add of exp * exp
  | Mul of exp * exp
  | GT of exp * exp
  | If of exp * exp * exp
  | Let of id * exp * exp
  | Id of id

let is_value (e : exp) =
  match e with
  | Bool _ -> true
  | Num _ -> true
  | _ -> false
```

(a) Syntax.

```
let to_int (e : exp) : int = match e with
  | Num n -> n
  | _ -> failwith "expected Num"

let to_bool (e : exp) : bool = match b with
  | Bool b -> b
  | _ -> failwith "expected Bool"

let rec subst (x : id) (v : exp) (e : exp) : exp =
  match e with
  | Num n -> Num n
  | Bool b -> Bool b
  | Add (e1, e2) -> Add (subst x v e1, subst x v e2)
  | Mul (e1, e2) -> Mul (subst x v e1, subst x v e2)
  | GT (e1, e2) -> GT (subst x v e1, subst x v e2)
  | Let (y, e1, e2) ->
    Let (y, subst x v e1, if x = y then e2 else subst x v e2)
  | Id y -> if x = y then v else Id y

let rec eval (e : exp) : exp = match e with
  | Num n -> Num n
  | Bool b -> Bool b
  | Add (e1, e2) -> Num (to_int (eval e1) + to_int (eval e2))
  | Mul (e1, e2) -> Num (to_int (eval e1) * to_int (eval e2))
  | GT (e1, e2) -> Bool (to_int (eval e1) > to_int (eval e2))
  | If (e1, e2, e3) -> if to_bool (eval e1) then eval e2 else eval e3
  | Let (x, e1, e2) -> eval (subst x (eval e1) e2)
  | Id x -> failwith ("free identifier " ^ x)
```

(b) Semantics.

Figure 4.3: OCaml implementation of fig. 4.1

$\boxed{e}$

$$
\begin{aligned}
e &::= \cdots \\
&\mid\ e_1\ e_2 \\
&\mid\ \lambda x.e \\
v &::= \cdots \\
&\mid\ \lambda x.e
\end{aligned}
$$

(a) Syntax.

$\boxed{e \Downarrow v}$

$$\text{App}\ \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v \qquad e[x \mapsto v] \Downarrow v'}{e_1\ e_2 \Downarrow v'}$$

(b) Semantics.

$\boxed{e[x \mapsto v] = \mathrm{e}}$

$$
\begin{aligned}
&\cdots \\
e_1\ e_2[x \mapsto v] &= e_1[x \mapsto v]\ e_2[x \mapsto v] \\
\lambda x.e[x \mapsto v] &= \lambda x.e \\
\lambda y.e[x \mapsto v] &= \lambda y.e[x \mapsto v] && \text{when } x \neq y
\end{aligned}
$$

(c) Substitution.

Figure 4.4: First-class functions. Extends fig. 4.2.

4.2 Implementation

Figure 4.3 is an OCaml implementation of our language. You should study this definition and the inference rules in fig. 4.1 to ensure that they are in close correspondence.

5 Functions

In this section, we extend our language with *functions*, similar to functions in OCaml. The syntax and semantics of this extension are given in fig. 4.4. In this extension, functions are values, which means that they are values just like numbers and booleans. This allows functions to be passed to other functions, returned from functions, bound to identifiers, and so on. Therefore, these are *first-class functions*, as discussed last week.

An apparent shortcoming of this extension that all functions take exactly one argument. However, we can encode multi-argument functions by leveraging first-class functions. For example, suppose we want to write a function that takes two arguments x and y and calculates $10 * x * y$. Although we cannot write a two-argument function, we can do this:

$$\lambda x.\lambda y.10 * x * y$$

This is known as *currying*. It turns out OCaml uses this mechanism too. The following three OCaml definitions are equivalent:

```
let add1 x y = x + y

let add2 = fun x y -> x + y

let add3 = fun x -> fun y -> x + y
```