Expressions

- Basic language constructs for generating values.
- Given by a grammar:

\[
E \rightarrow E + E \\
E \rightarrow E - E \\
E \rightarrow E \times E \\
E \rightarrow - E \\
E \rightarrow (E) \\
E \rightarrow \text{id} \\
E \rightarrow \text{int\_const}
\]
Meaning of Expressions

- Meaning for expressions are given by “semantic functions” that associate a value with every expression.
  - What is the value of \( x + 1 \)?
  - What is the value of \( f(x) \) where \( f \) is defined as \( \text{int } f(\text{int } i) \{ \text{return } i+1; \} \)

  Depends on what the value of \( x \) is.

- An expression’s value can be determined when the values of all variables in that expression are given.

- How to represent values of variables?
  - **Environment**: maps variable name to locations
  - **Store**: maps locations to values

Example: C flat (C♭)

A small language to illustrate how semantic functions are written.

- **Values**
  - Integer constants
  - Boolean constants (true, false)

- **Variables of type**
  - int
  - Pointers

Expressions in C♭

\[
\begin{align*}
E & \to E \text{ arith\_op } E \\
E & \to - E \\
E & \to (E) \\
E & \to \text{id} \\
E & \to \text{int\_const} \\
\text{arith\_op} & \to + | - | * \\
C & \to E \text{ comp\_op } E \\
C & \to -E \\
C & \to (E) \\
C & \to \text{id} \\
C & \to \text{int\_const} \\
\text{comp\_op} & \to == | < \\
\text{logical\_op} & \to && | ||
\end{align*}
\]
Abstract Syntax of C

Expressions

```
type expr = Add of expr * expr  | Sub of expr * expr  | Mul of expr * expr  | Neg of expr  | Id of string  | IntConst of int;;
      |     |     |          |      |       |
      |     |     |          |      |       |
      |     |     |          |      |       |
      |     |     |          |      |       |
      |     |     |          |      |       |
      |     |     |          |      |       |
```

```
type cond = Equal of expr * expr  | Less of expr * expr  | And of cond * cond  | Or of cond * cond  | Not of cond  | True | False;;
      |     |     |          |       |       |       |
```

Abstract syntax of C (Continued)

- Each expression in concrete syntax can be represented by an equivalent expression in abstract syntax.
- Examples:
  - Concrete  Abstract
    - \( x+1 \)  \( \text{Add(} \text{Id("x"), IntConst(1))} \)
    - \( x*(y+3) \)  \( \text{Mul(} \text{Id("x"), Add(} \text{Id("y"), IntConst(3)))} \)
    - \( x == y \)  \( \text{Equal(} \text{Id("x"), Id("y"))} \)

- Abstract syntax ignores certain details (e.g., parenthesis in expressions), but makes certain features explicit (e.g. the “kind” of expression).

Environment and Store

- Only values we can store for now are integers.
  ```
type storable = Intval of integer;;
  When we add pointers to the languages, we will add to the definition of value.
  ```

- Locations can be simply represented by integers.
  ```
type location = int;;
  ```
Environment and Store

- Store maps locations to values.
  
  ```ocaml
type store = location * storable list;;
```

- Example: `[(1, Int(3)), (2, Int(7))]`: Location 1 has value 3 and 2 has value 7.

- Functions over store:
  - `value_at: store * location -> storable`

Environment maps variables to locations.

- `type environment = string * location list;;`

- Example: `[("x", 1), ("y", 2)]`: Variable x is at location 1 and y is at location 2.

- Functions over environment:
  - `binding_of: environment * string -> location`

The meaning of expressions

- What is the value of `x + 1`?
  - It is the value of `x` added to the value of 1.
  - The value of `x` is given by
    - `the environment` which specifies the location associated with `x`, and
    - `the store` which specifies the values stored in locations.

- “Value of” can be viewed as a function
  - `eval_expr: expr * environment * store -> value`

Expression evaluation

- Order of evaluation

- For the abstract syntax tree

```
   +
  /   
+   5
 / 
+   +
 / 
x   
3   2
```

- the equivalent expression is `(x + 3) + (2 + 4) + 5`
One possible semantics:
- evaluate AST bottom-up, left-to-right.
This constrains optimization that uses mathematical properties of operators
- (e.g. commutativity and associativity)
- e.g., it may be preferable to evaluate of $e_1 + (e_2 + e_3)$ instead of $(e_1 + e_2) + e_3$
- $(x+0) + (y+3) + (z+4) = x+y+z+0+3+4 = x+y+z+7$
the compiler can evaluate $0+3+4$ at compile time, so that at runtime, we have two fewer addition operations.

Some languages leave order of evaluation unspecified.
- even the order of evaluation of procedure parameters are not specified.

Problem:
- Semantics of expressions with side-effects, e.g., $(x++) + x$
  - If initial value of $x$ is 5
    - left-to-right evaluation yields 11 as answer, but
    - right-to-left evaluation yields 10
- So, languages with expressions with side-effects forced to specify evaluation order
- Still, a bad programming practice to use expressions where different orders of evaluation can lead to different results
  - Impacts readability (and maintainability) of programs

Left-to-right evaluation

Left-to-right evaluation with short-circuit semantics is appropriate for boolean expressions.
- $e_1 && e_2$: $e_2$ is evaluated only if $e_1$ evaluates to true.
- $e_1 || e_2$: $e_2$ is evaluated only if $e_1$ evaluates to false.

This semantics is convenient in programming:
- Consider the statement: if ($(i<n) && a[i]! = 0$)
- With short-circuit evaluation, $a[i]$ is never accessed if $i >= n$
- Another example: if ($(p!=NULL) && p->value>0$)
Left-to-right evaluation (Continued)

- **Disadvantage:**
  - In an expression like “if((a==b)||(c=d))”
  - The second expression has a statement. The value of c may or may not be the value of d, depending on if a == b is true or not.

- **Bottom-up:**
  - No order specified among unrelated subexpressions.
  - Short-circuit evaluation of boolean expressions.

- **Delayed evaluation**
  - Delay evaluation of an expressions until its value is absolutely needed.
  - Generalization of short-circuit evaluation.

---

Evaluating expressions

Assume that we are interested only in int values:

```
 eval_expr: expr * environment * store -> int
```

Recall:

| type expr = Add of expr * expr | type location = int;|
| Sub of expr * expr | type storable = |
| Mul of expr * expr | Intval of integer;|
| Neg of expr | type store = |
| Id of string | location * storable list;|
| IntConst of int ;| type environment = |

```
eval_expr(Id(x), env, store) = i
    where binding_of(env, x) = l
    and value_at(store, l) = Intval(i)
```

---

Evaluating expressions: The Program

```
eval_expr(expr, env, store) =
    match expr with
    | IntConst(i) -> i
    | Id(x) ->
        let l = binding_of(env, x)
        in let Intval(i) = value_at(store, l)
        in i
    | Add(e1, e2) ->
        let v1 = eval_expr(e1, env, store)
        and v2 = eval_expr(e2, env, store)
        in v1 + v2
    ...
```

Similarly we can define `eval_cond: cond * environment * store -> bool`
Consider evaluating conditions with the following fragment:

\[
\text{Or}(c_1, c_2) \rightarrow \\
\text{let } b_1 = \text{eval}\_\text{cond}(c_1, \text{env}, \text{store}) \\
\text{and } b_2 = \text{eval}\_\text{cond}(c_2, \text{env}, \text{store}) \\
\text{in } b_1 \text{ || } b_2
\]

What is the effect of \(i==0\) \text{ || } (x/i)\?

**Short-circuit evaluation:** For \(c_1 \text{ || } c_2\), evaluate \(c_2\) only if \(c_1\) is false.

\[
\text{Or}(c_1, c_2) \rightarrow \\
\text{if } (\text{eval}\_\text{cond}(c_1, \text{env}, \text{store})) \\
\text{then true} \\
\text{else eval}\_\text{cond}(c_2, \text{env}, \text{store})
\]

In the fragment of C considered so far, expressions do not have any side effect (i.e. cannot change the store) and hence, order of evaluation does not change the final result.

In C/C++/Java/..., expressions may have side effects (e.g. \(x++\))

Side effects modify the store

Expression valuation function then becomes:

\[
\text{eval}\_\text{expr}: \text{expr} \times \text{environment} \times \text{store} \rightarrow (\text{int} \times \text{store})
\]

i.e., meaning that the expression returns its value *and the updated store*