The Simple Imperative Language

Compiler Construction
Dan R. Ghica

evaluation

- key formal definition of a programming language
- inductive on the syntax tree
- ‘reduction’
- “operational semantics”
  - “big step”

expressions

- The set of values v ∈ {…,-2,-1,0,1,2,…}
- If e ⟶ v and e’ ⟶ v’
  then e+e’⟶v’’ where v’’=v+v’
- If e ⟶ v and e’ ⟶ v’
  then e*e’⟶v’’ where v’’=v×v’
- etc.

example expressions

- 7+3⟶10 and 10*5⟶50
- so (7+3)*5⟶50

- 5-3⟶2
- 4-1⟶3
  - 3*2⟶6
    - (5-3)*(4-1)⟶6

implementing expressions

```
let rec eval_exp = function
  | Operator (Plus, e1, e2)
     → eval_exp e1 + eval_exp e2
  | Operator (Minus, e1, e2)
     → eval_exp e1 - eval_exp e2
  …
  | Const n → n

# eval_exp (Operator(Plus, Const 7, Const 3));;
- : int = 10
```

imperative code

- we need an additional parameter: **store**
- the store is a function from **locations** to **values**
  - **values** can be just data (at first…)
  - **global** variables (at first…)

- if (s, e)⟶(s’, v’) and (s’, e’)⟶(s’’, v’’)
  then (s, e := e’) ⟶ (**update** (s’, (v’, v’’)), v)

- if (s, e)⟶(s’, v’) then (s, !e)⟶(s’, **retrieve**(s’, v’))

imperative code

- if (s, e)⟶(s’, v’) and (s’, e’)⟶(s’’, v’’)
  then (s, e;e’) ⟶ (s’’, v’’)

- if (s, e)⟶(s’, 0)
  then (s, while e do e’)⟶(s’, 0)

- if (s, e)⟶(s’,n) and (s’, e’; while e do e’)⟶(s’’, 0)
  then (s, while e do e’)⟶(s’’, 0)

implementing imperative code (hacky!)

```
let store = Hashtbl.create 100

…
   | Asg (Identifier x, e2) →
     let v2 = eval_exp e2 in (* may update store *)
     Hashtbl.replace store x v2; v2
 | Seq (e1, e2) →
     let _  = eval_exp e1 in (* may update store *)
     let v2 = eval_exp e2 in (* may update store *)
     v2
 | Identifier x →   (* implicit dereference 😐*)
     Hashtbl.find store x
```