Prove the Correctness of a Compiler

- The number of bugs in compilers is very small compared to those in other programs.
- But you have Thompson attacks.
- Also we want to verify program sources, but run machine code.

Last Week

- specified which programs we can write (syntax)
- (partly) specified what a program means (semantics)

Syntax (1)

```
datatype aexp =
   N nat
 | X loc
 | Op1 "nat ⇒ nat" aexp
 | Op2 "nat ⇒ nat ⇒ nat" aexp aexp

datatype bexp =
   TRUE
 | FALSE
 | ROp  "nat ⇒ nat ⇒ bool" aexp aexp
 | NOT bexp
 | AND bexp bexp
 | OR  bexp bexp
```

Syntax (2)

```
datatype cmd =
   SKIP
  | ASSIGN loc aexp         ("_ ::= _ " 60)
  | SEQ   cmd cmd           ("_; _" [60, 60] 10)
  | COND   bexp cmd cmd     ("IF _ THEN _ ELSE _" 60)
  | WHILE  bexp cmd         ("WHILE _ DO _" 60)
```

- What does the factorial program look like?
- What is SKIP useful for?

Semantics of Aexps

$$\frac{}{(\text{N n,m}) \longrightarrow_a \text{n}} \qquad \frac{}{(\text{X i,m}) \longrightarrow_a \text{m i}}$$

$$\frac{(\text{e,m}) \longrightarrow_a \text{n}}{(\text{Op1 f e,m}) \longrightarrow_a \text{f n}}$$

$$\frac{(\text{e0,m}) \longrightarrow_a \text{n0} \qquad (\text{e1,m}) \longrightarrow_a \text{n1}}{(\text{Op2 f e0 e1,m}) \longrightarrow_a \text{f n0 n1}}$$

- What is the semantics of bexps?

Semantics of Bexps

$$\frac{}{(\mathsf{TRUE},m) \longrightarrow_b \mathsf{True}} \qquad \frac{}{(\mathsf{FALSE},m) \longrightarrow_b \mathsf{False}}$$

$$\frac{(e1,m) \longrightarrow_a n1 \qquad (e2,m) \longrightarrow_a n2}{(\mathsf{ROp}\ f\ e1\ e2,m) \longrightarrow_b f\ n1\ n2}$$

$$\frac{(e,m) \longrightarrow_b b}{(\mathsf{NOT}\ e,m) \longrightarrow_b \neg\ b}$$

$$\frac{(e1,m) \longrightarrow_b b1 \qquad (e2,m) \longrightarrow_b b2}{(\mathsf{AND}\ e1\ e2,m) \longrightarrow_b b1 \wedge b2}$$

$$\frac{(e1,m) \longrightarrow_b b1 \qquad (e2,m) \longrightarrow_b b2}{(\mathsf{OR}\ e1\ e2,m) \longrightarrow_b b1 \vee b2}$$

$$\frac{}{(\mathrm{SKIP}, m) \longrightarrow_c m} \qquad \frac{(a, m) \longrightarrow_a n}{(x ::= a\ , m) \longrightarrow_c m(x := n)}$$

$$\frac{(c0, m) \longrightarrow_c m' \qquad (c1, m') \longrightarrow_c m''}{(c0;\ c1, m) \longrightarrow_c m''}$$

$$\frac{(e, m) \longrightarrow_b \mathrm{True} \qquad (c0, m) \longrightarrow_c m'}{(\mathrm{IF}\ e\ \mathrm{THEN}\ c0\ \mathrm{ELSE}\ c1, m) \longrightarrow_c m'}$$

$$\frac{(e, m) \longrightarrow_b \mathrm{False} \qquad (c1, m) \longrightarrow_c m'}{(\mathrm{IF}\ e\ \mathrm{THEN}\ c0\ \mathrm{ELSE}\ c1, m) \longrightarrow_c m'}$$

$$\frac{(e, m) \longrightarrow_b \mathrm{False}}{(\mathrm{WHILE}\ e\ \mathrm{DO}\ c, m) \longrightarrow_c m}$$

$$\frac{\begin{array}{c}(e, m) \longrightarrow_b \mathrm{True} \\ (c, m) \longrightarrow_c m' \qquad (\mathrm{WHILE}\ e\ \mathrm{DO}\ c, m') \longrightarrow_c m''\end{array}}{(\mathrm{WHILE}\ e\ \mathrm{DO}\ c, m) \longrightarrow_c m''}$$

Equivalence

- Two commands are equivalent

  $$c \approx c' \equiv \forall\, m\; m'.\; (c,m) \longrightarrow_c m' = (c',m) \longrightarrow_c m'$$

Equivalence

- Two commands are equivalent

  $$c \approx c' \equiv \forall\, m\ m'.\ (c,m) \longrightarrow_c m' = (c',m) \longrightarrow_c m'$$

- $SKIP;\ SKIP \approx SKIP$

Instructions

- We have a memory, a stack and a single register.

```
datatype instr =
   JMPF  "nat"                  jump forward n steps, if reg. is False
 | JMPB  "nat"                        backward n steps
 | FETCH "loc"                        move memory to top of stack
 | STORE "loc"                        pop top of stack to memory
 | PUSH  "nat"                        push to stack
 | POP                                stack to register
 | SET   "nat"                        set register
 | OPU   "nat ⇒ nat"                  pop one from stack and apply f
 | OPB   "nat ⇒ nat ⇒ nat"            pop two from stack and apply f
```

Booleans

- We encode booleans as 0 and 1.

```
fun
  WRAP::"bool⇒nat"
where
  "WRAP True = 1"
| "WRAP False = 0"

fun
  MNot::"nat ⇒ nat"
where
  "MNot 0 = 1"
| "MNot (Suc 0) = 0"
```

Compiler for Aexps

```
fun
 compa
where
  "compa (N n) = [PUSH n]"
| "compa (X l) = [FETCH l]"
| "compa (Op1 f e) = (compa e) @ [OPU f]"
| "compa (Op2 f e1 e2) =
        (compa e1) @ (compa e2) @ [OPB f]"
```

Compiler for Bexps

```
fun
 compb
where
  "compb (TRUE) = [PUSH 1]"
| "compb (FALSE) = [PUSH 0]"
| "compb (ROp f e1 e2) = (compa e1) @ (compa e2)
                 @ [OPB (λx y. WRAP (f x y))]"
| "compb (NOT e) = (compb e) @ [OPU MNot]"
| "compb (AND e1 e2) =
    (compb e1) @ (compb e2) @ [OPB MAnd]"
| "compb (OR e1 e2) =
    (compb e1) @ (compb e2) @ [OPB MOr]"
```

```
fun
  compc :: "cmd ⇒ instr list"
where
  "compc SKIP = []"
| "compc (x::=a) = (compa a) @ [STORE x]"
| "compc (c1;c2) = compc c1 @ compc c2"
| "compc (IF b THEN c1 ELSE c2) =
    (compb b) @ [POP] @
    [JMPF (length(compc c1) + 1)] @ compc c1 @
    [SET 0, JMPF (length(compc c2))] @ compc c2"
| "compc (WHILE b DO c) =
    (compb b) @ [POP] @
    [JMPF (length(compc c) + 1)] @ compc c @
    [JMPB (length(compc c)+length(compb b)+2)]"
```

Compiler Lemma for Aexps

- We have to know how machine programs are executed.

```
types instrs = "instr list"
types stack = "nat list"

inductive
 step ("'(_,_,_') ⟶m '(_,_,_')")
where
  "(PUSH n#p, s, m) ⟶m (p, n#s, m)"
| "(FETCH l#p, s, m) ⟶m (p, m l#s, m)"
| "(OPU f#p, n#s, m) ⟶m (p, f n#s, m)"
| "(OPB f#p, n1#n2#s, m) ⟶m (p, f n2 n1#s, m)"
```

Many Transitions

- We have to build the transitive closure.

Many Transitions

- We have to build the transitive closure.

$$\frac{}{(p,s,m) \longrightarrow_{m^*} (p,s,m)}$$

$$\frac{(p1,s1,m) \longrightarrow_{m} (p2,s2,m)}{(p1,s1,m) \longrightarrow_{m^*} (p2,s2,m)}$$

$$\frac{(p1,s1,m) \longrightarrow_{m^*} (p2,s2,m) \quad (p2,s2,m) \longrightarrow_{m^*} (p3,s3,m)}{(p1,s1,m) \longrightarrow_{m^*} (p3,s3,m)}$$

Compiler Lemma for Aexps

- If (e,m) $\longrightarrow$a n, then
  (compa e,[],m) $\longrightarrow$m* ([],[n],m)

Compiler Lemma for Aexps

- If (e,m) $\longrightarrow_a$ n, then
  (compa e,[],m) $\longrightarrow_m^*$ ([],[n],m)

- For all s, if (e,m) $\longrightarrow_a$ n, then
  (compa e,s,m) $\longrightarrow_m^*$ ([],n # s,m)

Compiler Lemma for Aexps

- If (e,m) ⟶a n, then
  (compa e,[],m) ⟶m* ([],[n],m)

- For all s, if (e,m) ⟶a n, then
  (compa e,s,m) ⟶m* ([],n # s,m)

```
lemma append:
  assumes a: "(p1,s,m) ⟶m (p2,s',m')"
  shows "(p1@p3,s,m) ⟶m (p2@p3,s',m')"
using a by (induct) (auto intro: step.intros)

lemma appends:
  assumes a: "(p1,s,m) ⟶m* (p2,s',m')"
  shows "(p1@p3,s,m) ⟶m* (p2@p3,s',m')"
using a by (induct) (auto intro: steps.intros append)
```

Compiler Lemma

- If $(e,m) \longrightarrow_a n$, then

  $$\forall s.\ (\text{compa } e, s, m) \longrightarrow_{m^*} ([], n \# s, m)$$

Compiler Lemma

- If $(e,m) \longrightarrow_a n$, then

  $\forall s.\ (\text{compa}\ e,s,m) \longrightarrow_{m^*} ([],n \# s,m)$

- If $(e,m) \longrightarrow_b b$, then

  $\forall s.\ (\text{compb}\ e,s,m) \longrightarrow_{m^*} ([],\text{WRAP}\ b \# s,m)$

```
inductive
  step' ("'(_,_,_,_,_') ⟶m '(_,_,_,_,_')")
where
  "(PUSH n#p, q, r, s, m) ⟶m (p, PUSH n#q, r, n#s, m)"
| "(FETCH l#p, q, r, s, m) ⟶m (p, FETCH l#q, r, m l#s, m)"
| "(OPU f#p, q, r, n#s, m) ⟶m (p, OPU f#q, r, f n#s, m)"
| "(OPB f#p,q,r,n1#n2#s,m) ⟶m (p, OPB f#q,r,f n2 n1#s,m)"
| "(POP#p, q, r, n#s, m) ⟶m (p, POP#q, n, s, m)"
| "(SET n#p, q, r, s, m) ⟶m (p, SET n#q, n, s, m)"
| "(STORE x#p, q, r, n#s, m) ⟶m (p, STORE l#q, r, s, m(x:=n))"
| "(JMPF i#p, q, Suc 0, s, m) ⟶m (p, JMPF i#q, Suc 0, s, m)"
| "i≤length p⟹
  (JMPF i#p, q, 0, s, m) ⟶m
          (drop i p, (rev (take i p))@(JMPF i#q), 0, s, m)"
| "i≤length q⟹
   (JMPB i#p, q, r, s, m) ⟶m
          ((rev (take i q))@(JMPF i#p), drop i q, r, s, m)"
```

Points to Take Home

- If you want to show the correctness of a compiler, you have to specify precisely the language, compiler and machine.
- The proofs in the compiler lemma are mostly inductions.
- They are tedious, but cases are easily forgotten (therefore use a theorem prover).
- Proving the compiler lemma helps to debug the compiler.