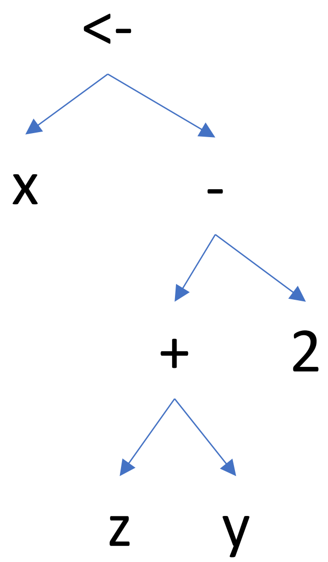
CS443: Compiler Construction

Lecture 6: Flattening Expressions, Basic Control Flow

Stefan Muller

Flattening expressions

```
x <- y + z – 2  → %temp = add i32 %y %z
                     %x = sub i32 %temp 2
```



One approach: destination passing

```
let rec compile_exp (dest: var) (e: exp) : inst list =
  match e with
  | ENum n -> [dest = set n]
  | EUnop (UNeg, e1) ->
     let dest1 = new_temp () in
     (compile_exp dest1 e1) @ [dest = sub 0 dest1]
   | EAssign (EVar v, e1) ->
     (compile_exp v e1) @ (* … need to copy v to dest *)
```

One approach: destination passing

```
let rec compile_exp (dest: var) (e: exp) : inst list =
  match e with
  | ENum n -> [dest = set n]
  | EUnop (UNeg, e1) ->
     let dest1 = new_temp () in
     (compile_exp dest1 e1) @ [dest = sub 0 dest1]
   | EAssign (EVar v, e1) ->
     (compile_exp v e1) @ (* … need to copy v to dest *)
```

```
x <- y + z – 2
```

```
temp1 = set y
temp2 = set z
temp3 = add temp1 temp2
temp4 = set 2
x = sub temp3 temp4
```

Another approach

```
let rec compile_exp (e: exp) : inst list * value =
  match e with
  | ENum n -> [], n
  | EBinop (BAdd, e1, e2) ->
     let (is1, v1) = compile_exp e1 in
     let (is2, v2) = compile_exp e2 in
     let d = new_temp () in
     (is1 @ is2 @ [d = add v1 v2], d)
   | EAssign (EVar v, e1) ->
     let (is, d) = compile_exp e1 in
             (is @ [v = set d], v)
```

`set` isn't an LLVM instruction!
One way to do it:
`%v = bitcast <ty> %d to <ty>`

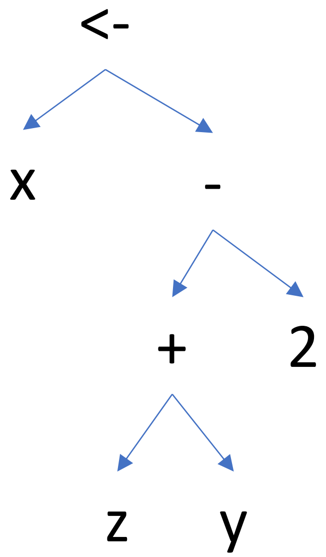
Another approach

```
let rec compile_exp (e: exp) : inst list * value =
  match e with
  | ENum n -> [], n
  | EBinop (BAdd, e1, e2) ->
     let (is1, v1) = compile_exp e1 in
     let (is2, v2) = compile_exp e2 in
     let d = new_temp () in
     (is1 @ is2 @ [d = add v1 v2], d)
   | EAssign (EVar v, e1) ->
     let (is, d) = compile_exp e1 in
             (is @ [v = set d], v)
```

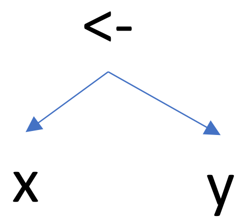
```
temp1 = add y z
temp2 = sub temp1 2
x = set temp2
```

```
x <- y + z – 2
```

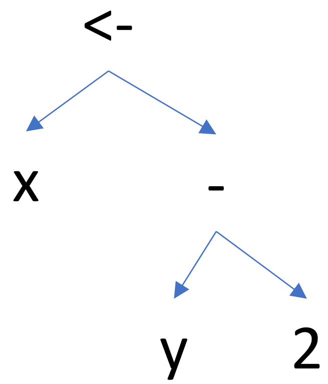
A somewhat better approach: Maximal Munch



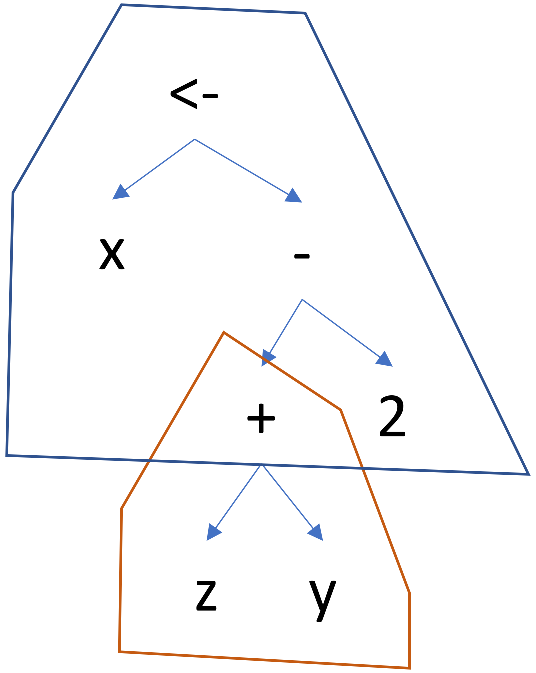
```
x = set y
```



```
x = sub y 2
```



A somewhat better approach: Maximal Munch



```
temp1 = add z y
x = sub temp1 2
```

(In practice, doesn’t matter a lot)

Boolean operators should short-circuit!

```
LOGICAL RESULT
RESULT <- 1 > 0 OR 42 / 0 = 2
```

Idea: Compile Boolean expressions into code that jumps to one of two labels

```
compile_bexpr(bexpr : exp, tdest : label, fdest : label)

compile_bexpr(x < 10, tdest, fdest) =
  %temp = icmp lt i32 %x 10
  br i1 %temp, label tdest, label fdest
```

Idea: Compile Boolean expressions into code that jumps to one of two labels

```
compile_bexpr(bexpr : exp, tdest : label, fdest : label)

compile_bexpr(e1 AND e2, tdest, fdest) =
    compile_bexpr(e1, %e1true, fdest)
  e1true:
    compile_bexpr(e2, tdest, fdest)
```

short-circuit

Idea: Compile Boolean expressions into code that jumps to one of two labels

```
compile_bexpr(bexpr : exp, tdest : label, fdest : label)

compile_bexpr(e1 OR e2, tdest, fdest) =
    compile_bexpr(e1, tdest, %e1false)
  e1false:
    compile_bexpr(e2, tdest, fdest)
```

short-circuit

If/then compile to conditional jumps

```
IF x < 10 s1 ELSE s2
```

looks like output of compile_bexpr…

```
  %temp = icmp lt i32 %x 10
  br i1 %temp, label %label1, label %label2
label1:
  (Compilation of s1)
  br label %label3
label2:
  (Compilation of s2)
  br label %label3
label3: …
```

Potential code duplication issue: Boolean expressions should be able to *both* return a Boolean value *and* result in a jump

```
RESULT <- 1 > 0 OR 42 / 0 = 2
```

vs.

```
IF 1 > 0 OR 42 / 0 = 2 THEN … ELSE …
```

Approach 1

Compile expr

```
  e1 AND e2
| e1 OR e2  ->
 (Need to short circuit; at end:
  truedest:
    %dest = set 1
  falsedest
    %dest = set 0
)
| <arith ops> -> arith ops…
| <comparisons> -> Icmp
| <casts> ->
  (be careful casting to/from
   LOGICAL)
```

Compile branching expr

```
%dest = compile expr
ICondBr…
```

Approach 1

IF 1 > 0 OR 42 / 0 = 2 THEN … ELSE …

```
  %c1 = icmp sgt i32 1 0
  br i1 %c1, label %l2, label %l1
l1:
  %temp = …
  %dest = icmp eq i32 %temp 2
  br label %l3
l2:
  %dest = set 1
  br label %l3
l3:
   br i1 %dest, label %ltrue, label %lfalse
```

Approach 1

`RESULT <- 1 > 0 OR 42 / 0 = 2`

```
  %c1 = icmp sgt i32 1 0
  br i1 %c1, label %l2, label %l1
l1:
  %temp = …
  %dest = icmp eq i32 %temp 2
  br label %l3
l2:
  %dest = set 1
  br label %l3
l3:
```

Approach 2

Compile expr

```
Boolean expressions ->
  compile branching expr…
  truedest:
    %dest = set 1
  falsedest
    %dest = set 0
| Integer expressions -> …
```

Compile branching expr

```
| e1 AND e2
| e1 OR e2 ->
  (short circuiting from before)
| <comparison ops> ->
  ICmp; ICondBr
| <cast from INTEGER to LOGICAL> ->
  …
```

Approach 2

`RESULT <- 1 > 0 OR 42 / 0 = 2`

```
  %c1 = icmp sgt i32 1 0
  br i1 %c1, label %l2, label %l1
l1:
  %temp = …
  %c2 = icmp eq i32 %temp 2
  br i1 %c2, label %l2, label %l3
l2:
  %dest = set 1
  br label %l4
l3:
  %dest = set 0
  br label %l4
l4:
```

(Some optimizations possible in both methods)

(Also, optimizations will take care of these anyway)

While loops have a backward jump

```
WHILE x < 10 s1

testl:
  %temp = icmp lt i32 %x 10
  br i1 %temp, label %bodyl, label %donel
bodyl:
  (compilation of s1)
  br label %testl
donel:
```

Unconditional jump back to test (NOT start of body!)

Example

```
WHILE x = 0 OR 10 / x > 2    x <- x + 1
```

```
test:
  %temp1 = icmp eq i32 %x 0
  br i1 %temp1, label %body, label %xnezero
xnezero:
  %temp2 = sdiv 10 i32 %x
  %temp3 = icmp gt i32 %temp2 2
  br i1 %temp3, label %body, label %done
```

(Probably) Not SSA!

```
body:
  %x = add i32 %x 1
  br label %test
done:
```