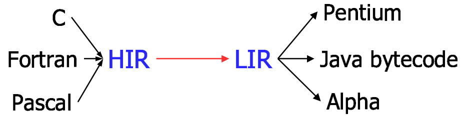
CS412/413

Introduction to Compilers
Radu Rugina

Lecture 15: Translating High IR to Low IR
22 Feb 02

Intermediate Representation

- Intermediate representation = internal representation
  - Is language-independent and machine-independent
- High IR: captures high-level language constructs
- Low IR: captures low-level machine features


C
Fortran
Pascal
HIR
LIR
Pentium
Java bytecode
Alpha

High-level IR

- Tree node structure very similar to the AST
- Contains high-level constructs common to many languages
  - Expression nodes
  - Statement nodes
- Expression nodes for:
  - Integers and program variables
  - Binary operations: e1 OP e2
    - Arithmetic operations
    - Logic operations
    - Comparisons
  - Unary operations: OP e
  - Array accesses: e1[e2]

High-level IR

- Statement nodes:
  - Block statements (statement sequences): (s1, …, sN)
  - Variable assignments: v = e
  - Array assignments: e1[e2] = e3
  - If-then-else statements: if c then s1 else s2
  - If-then statements: if c then s
  - While loops: while (c) s
  - Function call statements: f(e1, …, eN)
  - Return statements: return or return e
- May also contain:
  - For loop statements: for(v = e1 to e2) s
  - Break and continue statements
  - Switch statements: switch(e) { v1: s1, …, vN: sN }

High-level IR

- Statements may be expressions
- Statement expression nodes:
  - Block statements: (s1, …, sN)
  - Variable assignments: v = e
  - Array assignments: e1[e2] = e3
  - If-then-else statements: if c then s1 else s2
  - Function calls: f(e1, …, eN)
- There is a high IR node for each of the above.
  - All AST nodes are translated into the above IR nodes

Low-level IR

- Represents a set of instructions which emulates an abstract machine
- Arithmetic and logic instructions:
  - Binary : a = b OP c
    - Arithmetic operations
    - Logic operations
    - Comparisons
  - Unary operations: a = OP b
- Data movement instructions:
  - Copy : a = b
  - Load : a = [b] (load in a the value at address b)
  - Store: [a] = b (store at address a the value b)

Low-level IR

- Function call instructions:
  - Call instruction: call f(a1, …, aN)
  - Call assignment: a = call f(a1, …, aN)
  - Return instruction: return
  - Value return: return a
- Branch instructions:
  - Unconditional jump: jump L
  - Conditional jump: cjump c L
- These instructions are also called quadruples or three-address instructions

Translating High IR to Low IR

- We need to translate each high-level IR node to a low-level IR sequence of instructions
  - Expressions nodes: arithmetic, logic, comparison, unary, etc.
  - Statements nodes: blocks, if-then-else, if-then, while, function calls, etc.
  - Expression statements nodes: if-then-else, calls, etc.

Notation

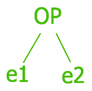
- Use the following notation:
  ⟦e⟧ = the low-level IR representation of high-level IR construct e
- ⟦e⟧ is a sequence of Low-level IR instructions
- If e is an expression (or a statement expression), it represents a value
- Denote by t = ⟦e⟧ the low-level IR representation of e, whose result value is stored in t
- For variable v: t = ⟦v⟧ is the copy instruction t = v

Translating Expressions

- Binary operations: t = ⟦ e1 OP e2 ⟧

  (arithmetic operations and comparisons)

  t1 = ⟦ e1 ⟧
  t2 = ⟦ e2 ⟧
  t = t1 OP t2



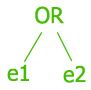
- Unary operations: t = ⟦ OP e ⟧

  t1 = ⟦ e ⟧
  t = OP t1

Translating Boolean Expressions

- t = ⟦ e1 OR e2 ⟧

  t1 = ⟦ e1 ⟧
  t2 = ⟦ e2 ⟧
  t = t1 OR t2

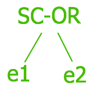


- … how about short-circuit OR?
- Should compute e2 only if e1 evaluates to false

Translating Short-Circuit OR

- Short-circuit OR: t = ⟦ e1 SC-OR e2 ⟧

  t = ⟦ e1 ⟧
  cjump t Lend
  t = ⟦ e2 ⟧
  label Lend

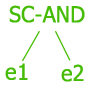


- … how about short-circuit AND?

Translating Short-Circuit AND

- Short-circuit AND: t = ⟦ e1 SC-AND e2 ⟧

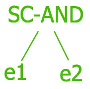
```
t = ⟦ e1 ⟧
cjump t Lnext
jump Lend
label Lnext
t = ⟦ e2 ⟧
label Lend
```



Another Translation

- Short-circuit AND: t = ⟦ e1 SC-AND e2 ⟧

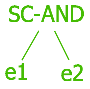
```
t1 = ⟦ e1 ⟧
t2 = not t1
cjump t2 Lend
t = ⟦ e2 ⟧
label Lend
```



Yet Another Translation

- Use another low-level IR: abstract machine with two kinds of conditional jumps
  - tjump c L : jump to L if c is true
  - fjump c L : jump to L if c is false
- Short-circuit AND: t = ⟦ e1 SC-AND e2 ⟧

```
t = ⟦ e1 ⟧
fjump t2 Lend
t = ⟦ e2 ⟧
label Lend
```

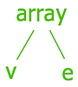


Translating Array Accesses

- Array access: t = ⟦ v[e] ⟧

(type of e1 is array[T] and S = size of T)

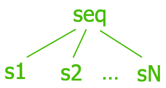
```
t1 = addr v
t2 = ⟦ e ⟧
t3 = t2 * S
t4 = t1 + t3
t = [t4]
```



Translating Statements

- Statement sequence: ⟦ s1; s2; …; sN ⟧

```
⟦ s1 ⟧
⟦ s2 ⟧
…
⟦ sN ⟧
```



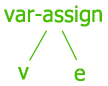
- IR instructions of a statement sequence = concatenation of IR instructions of statements

---

Assignment Statements

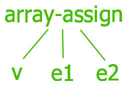
- Variable assignment: ⟦ v = e ⟧

```
v = ⟦ e ⟧
```



- Array assignment: ⟦ v[e1] = e2 ⟧

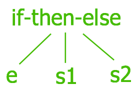
```
t1 = addr v
t2 = ⟦ e1 ⟧
t3 = t2 * S
t4 = t1 + t3
t5 = ⟦ e2 ⟧
[t4] = t5
```



Translating If-Then-Else

- ⟦ if (e) then s1 else s2 ⟧

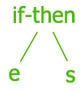
t1 = ⟦ e ⟧
fjump t1 Lfalse
⟦ s1 ⟧
jump Lend
label Lfalse
⟦ s2 ⟧
label Lend



Translating If-Then

- ⟦ if (e) then s ⟧

t1 = ⟦ e ⟧
fjump t1 Lend
⟦ s ⟧
label Lend



While Statements

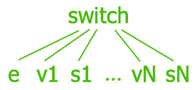
- ⟦ while (e) { s } ⟧

label Ltest
t1 = ⟦ e ⟧
fjump t1 Lend
⟦ s ⟧
jump Ltest
label Lend

Switch Statements

- ⟦ switch (e) { case v1: s1, …, case vN: sN } ⟧

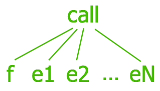
t = ⟦ e ⟧
c = t != v1
tjump c L2
⟦ s1 ⟧
jump Lend
label L2
c = t != v2
tjump c L3
⟦ s2 ⟧
jump Lend
…
label LN
c = t != vN
tjump c Lend
⟦ sN ⟧
label Lend



Call and Return Statements

- ⟦ call f(e1, e2, …, eN) ⟧

t1 = ⟦ e1 ⟧
t2 = ⟦ e2 ⟧
…
tN = ⟦ eN ⟧
call f(t1, t2, …, tN)



- ⟦ return e ⟧

t = ⟦ e ⟧
return t

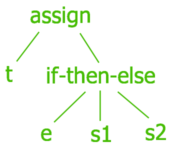
Statement Expressions

- So far: statements which do not return values
- Easy extensions for statement expressions:
  - Block statements
  - If-then-else
  - Assignment statements
- t = ⟦ s ⟧ is the sequence of low IR code for statement s, whose result is stored in t

Statement Expressions

- t = ⟦ if (e) then s1 else s2 ⟧

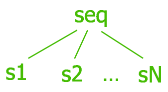
t1 = ⟦ e ⟧
cjump t1 Ltrue
t = ⟦ s2 ⟧
jump Lend
label Ltrue
t = ⟦ s1 ⟧
label Lend



Block Statements

- t = ⟦ s1; s2; …; sN ⟧

⟦ s1 ⟧
⟦ s2 ⟧
…
t = ⟦ sN ⟧



- Result value of a block statement = value of last statement in the sequence

Assignment Statements

- t = ⟦ v = e ⟧

v = ⟦ e ⟧
t = v

- Result value of an assignment statement = value of the assigned expression

Nested Expressions

- In these translations, expressions may be nested;
- Translation recurses on the expression structure
- Example: t = ⟦ (a - b) * (c + d) ⟧

t1 = a
t2 = b } ⟦ (a - b) ⟧
t3 = t1 – t2
t4 = b } ⟦ (a - b) * (c + d) ⟧
t5 = c
t5 = t4 + t5 } ⟦ (c + d) ⟧
t = t3 * t5

Nested Statements

- Same for statements: recursive translation
- Example: ⟦ if c then if d then a = b ⟧

t1 = c
fjump t1 Lend1
t2 = d
fjump t2 Lend2 } ⟦ if c then … ⟧
t3 = b } ⟦ a = b ⟧ } ⟦ if d … ⟧
a = t3
label Lend2
label Lend1

Issues

- These translations are straightforward
- … and inefficient:
  - May generate many temporary variables
  - May generate many labels
- Can optimize translation process:
  - Don't create temporaries for variables
  - Reuse temporary variables
  - Merge adjacent labels

Optimize Translation

- Example: t = ⟦ (a - b) * (c + d) ⟧

```
t1 = a
t2 = b
t3 = t1 – t2
t4 = b
t5 = c
t5 = t4 + t5
t = t3 * t5
```

⇨

```
t = a - b
t1 = b + c
t = t * t1
```