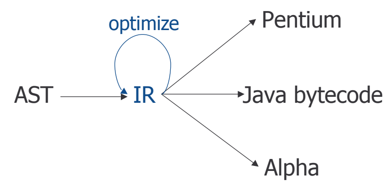
CS412/413

Introduction to Compilers
Radu Rugina

Lecture 17: From High IR to Low IR
26 Feb 03

Intermediate Code

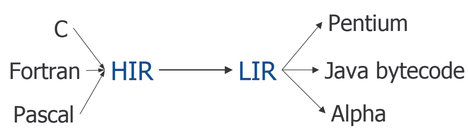
- IR = Intermediate Representation
- Allows language-independent, machine-independent optimizations and transformations



Multiple IRs

- Usually two IRs:

High-level IR	Low-level IR
Language-independent (but closer to language)	Machine independent (but closer to machine)



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

- Low-level representation is essentially an instruction set for an abstract machine
- Alternatives for low-level IR:
  - Three-address code or quadruples (Dragon Book):
    a = b OP c
  - Tree representation (Tiger Book)
  - Stack machine (like Java bytecode)

Three-Address Code

- In this class: three-address code
  a = b OP c
- Has at most three addresses (may have fewer)
- Also named quadruples because can be represented as: (a, b, c, OP)
- Example:
  a = (b+c)*(-e);
  t1 = b + c
  t2 = – e
  a = t1 * t2

Low IR Instructions

- Assignment instructions:
  - Binary operations: a = b OP c
    - arithmetic: ADD, SUB, MUL, DIV, MOD
    - logic: AND, OR, XOR
    - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - Unary operation a = OP b
    - Arithmetic MINUS or logic NEG
  - Copy instruction: a = b
  - Load /store: a = *b, *a = b
  - Other data movement instructions

Low IR Instructions (Ctd)

- Flow of control instructions:
  - label L : label instruction
  - jump L : Unconditional jump
  - cjump a L : conditional jump
- Function call
  - call f(a_1, …., a_n)
  - a = call f(a_1, …., a_n)
  - Is an extension to quads
- … IR describes the Instruction Set of an abstract machine

Temporary Variables

- The operands in the quadruples can be:
  - Program variables
  - Integer constants
  - Temporary variables
- Temporary variables = new locations
  - Use temporary variables to store intermediate values

Arithmetic / Logic Instructions

- Abstract machine supports a variety of different operations

  a = b OP c   a = OP b
- Arithmetic operations: ADD, SUB, DIV, MUL
- Logic operations: AND, OR, XOR
- Comparisons: EQ, NEQ, LE, LEQ, GE, GEQ
- Unary operations: MINUS, NEG

Data Movement

- Copy instruction: a = b
- Load/store instructions:

  a = *b   *a = b

  – Models a load/store machine
- Address-of instruction: a = &b
- Array accesses:

  a = b[i]   a[i] = b
- Field accesses:

  a = b.f   a.f = b

Branch Instructions

- Label instruction:

  label L
- Unconditional jump: go to statement after label L

  jump L
- Conditional jump: test condition variable a; if true, jump to label L

  cjump a L
- Alternative: two conditional jumps:

  tjump a L   fjump a L

Call Instruction

- Supports function call statements

  call f(a_1, …, a_n)
- … and function call assignments:

  a = call f(a_1, …, a_n)
- No explicit representation of argument passing, stack frame setup, etc.

Example

```
n = 0;
while (n < 10) {
   n = n + 1
}
```

```
n = 0
label test
t2 = n < 10
fjump t3 end
n = n + 1
jump test
label end
```


Another Example

```
m = 0;
if (c  == 0) {
   m = m+ n*n;
} else {
   m = m + n;
}
```

```
m = 0
t1 = c == 0
tjump t1 trueb
m = m+n
jump end
label trueb
t2 = n * n
m = m + t2
label end
```


How To Translate?

- May have nested language constructs
  – Nested if and while statements
- Need an algorithmic way to translate
- Solution:
  – Start from the AST representation
  – Define translation for each node in the AST
  – Recursively translate nodes in the AST

Notation

- Use the following notation:
  T[e] = the low-level IR representation of high-level IR construct e
- T[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 = T[e] the low-level IR representation of e, whose result value is stored in t
- For variable v: t = T[v] is the copy instruction t = v

Translating Expressions

- Binary operations: t = T[e1 OP e2]
  (arithmetic operations and comparisons)

  t1 = T[e1]
  t2 = T[e2]
  t = t1 OP t2

  OP
  e1 e2

- Unary operations: t = T[OP e]

  t1 = T[e]
  t = OP t1

  OP
  e

Translating Boolean Expressions

- t = T[e1 OR e2]

  t1 = T[e1]
  t2 = T[e2]
  t = t1 OR t2

  OR
  e1 e2

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

Translating Short-Circuit OR

- Short-circuit OR: t = T[e1 SC-OR e2]

  t = T[e1]
  tjump t Lend
  t = T[e2]
  label Lend

  SC-OR
  e1 e2

- … how about short-circuit AND?

Translating Short-Circuit AND

- Short-circuit AND: t = T[e1 SC-AND e2]

  t = T[e1]
  tjump t Lnext
  jump Lend
  label Lnext
  t = T[e2]
  label Lend

  SC-AND
  e1 e2

Another Translation

- Short-circuit AND: t = T[e1 SC-AND e2]

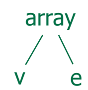
  t = T[e1]
  fjump t Lend
  t = T[e2]
  label Lend

  SC-AND
  e1 e2

Array and Field Accesses

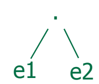
- Array access: t = T[v[e]]

  t1 = T[e]
  t = v[t1]



- Field access: t = T[e1.e2]

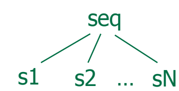
  t1 = T[e1]
  t2 = T[e2]
  t = t1.t2



Translating Statements

- Statement sequence: T[s1; s2; …; sN]

  T[s1]
  T[s2]
  …
  T[sN]

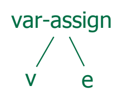


- IR instructions of a statement sequence = concatenation of IR instructions of statements

Assignment Statements

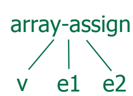
- Variable assignment: T[v = e]

  v = T[e]



- Array assignment: T[v[e1] = e2]

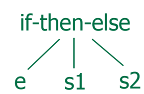
  t1 = T[e1]
  t2 = T[e2]
  v[t1] = t2



Translating If-Then-Else

- T[if (e) then s1 else s2]

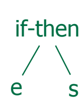
  t1 = T[e]
  fjump t1 Lfalse
  T[s1]
  jump Lend
  label Lfalse
  T[s2]
  label Lend



Translating If-Then

- T[if (e) then s]

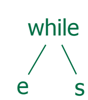
  t1 = T[e]
  fjump t1 Lend
  T[s]
  label Lend



While Statements

- T[while (e) { s }]

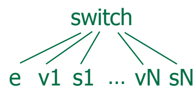
  label Ltest
  t1 = T[e]
  fjump t1 Lend
  T[s]
  jump Ltest
  label Lend



Switch Statements

- T[switch (e) { case v1: s1, ..., case vN: sN }]

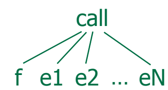
```
t = T[ e ]
c = t != v1
tjump c L2
T[ s1 ]
jump Lend
label L2
c = t != v2
tjump c L3
T[ s2 ]
jump Lend
...
label LN
c = t != vN
tjump c Lend
T[ sN ]
label Lend
```



Call and Return Statements

- T[call f(e1, e2, ..., eN)]

```
t1 = T[ e1 ]
t2 = T[ e2 ]
...
tN = T[ eN ]
call f(t1, t2, ..., tN)
```



- T[return e]

```
t = T[ e ]
return t
```

Statement Expressions

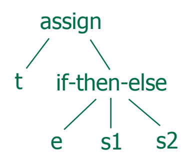
- So far: statements which do not return values
- Easy extensions for statement expressions:
  - Block statements
  - If-then-else
  - Assignment statements

- t = T[s] is the sequence of low IR code for statement s, whose result is stored in t

Statement Expressions

- t = T[if (e) then s1 else s2]

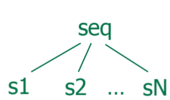
```
t1 = T[ e ]
cjump t1 Ltrue
t = T[ s2 ]
jump Lend
label Ltrue
t = T[ s1 ]
label Lend
```



Block Statements

- t = [s1; s2; ...; sN]

```
[ s1 ]
[ s2 ]
...
t = [ sN ]
```

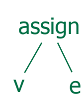


- Result value of a block statement = value of last statement in the sequence

Assignment Statements

- t = [v = e]

```
v = [ e ]
t = v
```



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