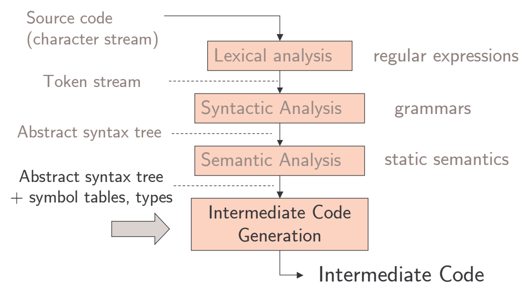
CS412/413

Introduction to Compilers
Radu Rugina

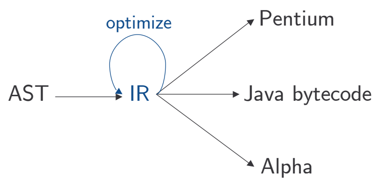
Lecture 16: Intermediate Representation
01 Mar 04

Where We Are



Intermediate Code

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



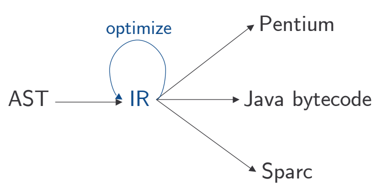
What Makes a Good IR?

- Easy to translate from AST
- Easy to translate to assembly
- Narrow interface: small number of node types (instructions)
  - Easy to optimize
  - Easy to retarget

AST (>40 node types)
↓
IR (13 node types)
↓
Pentium (>200 opcodes)

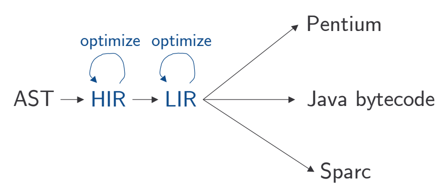
Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code



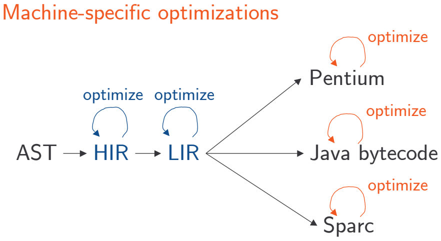
Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code
- Solution: use multiple IR stages



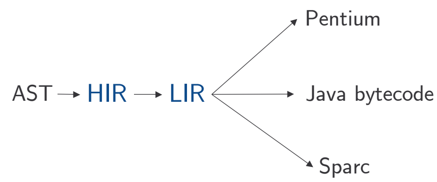
Machine Optimizations

- … some other optimizations take advantage of the features of the target machine
- Machine-specific optimizations



Next Lectures

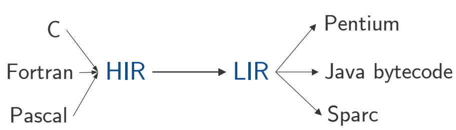
- Next few lectures: intermediate representation
- Optimizations covered later



Multiple IRs

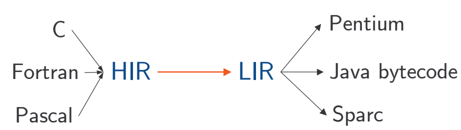
- Usually two IRs:

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



Multiple IRs

- Another benefit: a significant part of the translation from high-level to low-level is
  - Language-independent
  - Machine-independent



High-Level IR

- High-level intermediate representation is essentially the AST
  - Must be expressive for all input languages
- Preserves high-level language constructs
  - Structured control flow: if, while, for, switch, etc.
  - variables, methods
- Allows high-level optimizations
  - E.g, optimizations of nested "for" loops

Low-Level IR

- Low-level representation is essentially an abstract machine
- Has low-level constructs
  - Unstructured jumps, instructions
- Allows optimizations specific to these constructs (e.g. register allocation, branch prediction)

Low-Level IR

- Alternatives for low-level IR:
  - Three-address code or quadruples (Dragon Book):
    `a = b OP c`
  - Tree representation (Appel Book)
  - Mixed: three address for expressions and flat representation of control-flow
  - Stack machine (similar to Java bytecodes)
- Advantages:
  - Three-address code: easier dataflow analysis
  - Tree IR: easier instruction selection
  - Stack machine: better if the target has a stack model

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
```


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 (if language supports it):
  `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
t3 = not t2
cjump t3 end
label body
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
cjump 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
- Solution: syntax-directed translation
  - Start from the AST representation
  - Define translation for each node in the AST
  - Recursively translate nodes in the AST

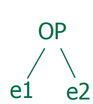
Notation

- Use the following notation:
  T[e] = the low-level IR code for high-level IR construct e
  - T[e] is a sequence of Low-level IR instructions
- If e is an expression , 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
- Temporary variables = new locations
  - Use temporary variables to store intermediate values during this translation

Translating Expressions

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

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



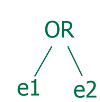
- Unary operations: `t = T[ OP e ]`

```
t1 := T[ e ]
t = OP t1
```

Translating Boolean Expressions

- `t := T[ e1 OR e2 ]`

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

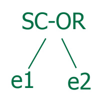


- … how about short-circuit OR?
- Should compute e2 only if e1 evaluates to false

Translating Short-Circuit OR

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

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

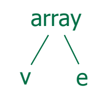


- … how about short-circuit AND?

Array and Field Accesses

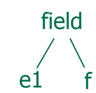
- Array access: `t := T[ v[e] ]`

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



- Field access: `t := T[ e1.f ]`

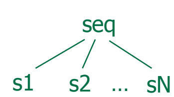
```
t1 := T[ e1 ]
t := t1.f
```



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

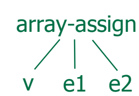
Assignment Statements

- Variable assignment: `T[ v = e ]`

```
v := T[ e ]
```

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

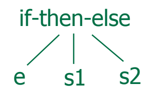
```
t1 := T[ e1 ]
t2 := T[ e2 ]
v[t1] = t2
```



Translating If-Then-Else

- `T[ if (e) { s1 } else { s2 } ]`

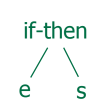
```
t1 := T[ e ]
fjump t1 Lfalse
T[ s1 ]
jump Lend
label Lfalse
T[ s2 ]
label Lend
```



Translating If-Then

- `T[ if (e) { s } ]`

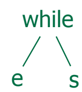
```
t1 := T[ e ]
fjump t1 Lend
T[ s ]
label Lend
```



While Statements

- T[while (e) { s }]

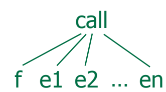
```
label Ltest
t1 := T[ e ]
fjump t1 Lend
T[ s ]
jump Ltest
label Lend
```



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
```