CS412/413

Introduction to Compilers

Radu Rugina

Lecture 14: Intermediate Representation

20 Feb 02

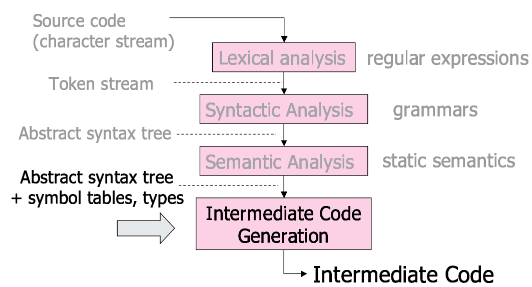
Semantic Analysis

- Check errors not detected by lexical or syntax analysis
- Scope errors:
  - Variables not defined
  - Multiple declarations
- Type errors:
  - Assignment of values of different types
  - Invocation of functions with different number of parameters or parameters of incorrect type
  - Incorrect use of return statements

Semantic Analysis

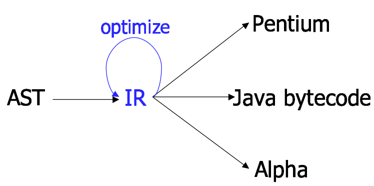
- Type checking
  - Use type checking rules
  - Static semantics = formal framework to specify type-checking rules
- There are also control flow errors:
  - Must verify that a break or continue statement is always enclosed by a while (or for) statement
  - Java: must verify that a break X statement is enclosed by a for loop with label X
  - Can easily check control-flow errors by recursively traversing the AST

Where We Are



Intermediate Code

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



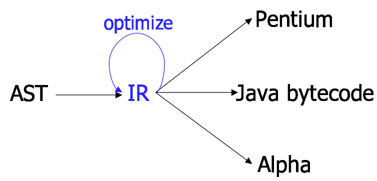
What Makes a Good IR?

- 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)

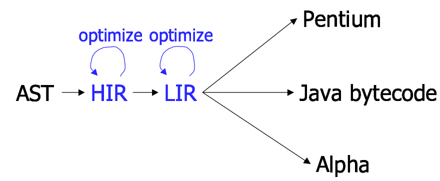
Multiple IRs

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



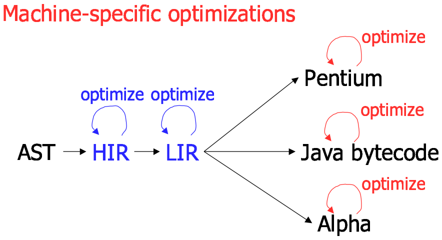
Multiple IRs

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



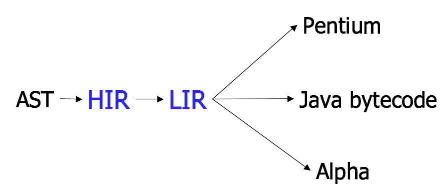
Machine Optimizations

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



Next Lectures

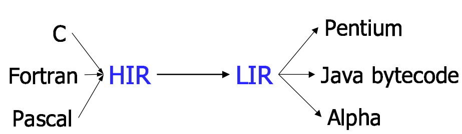
- Next few lectures: intermediate representation
- Optimizations covered later



Multiple IRs

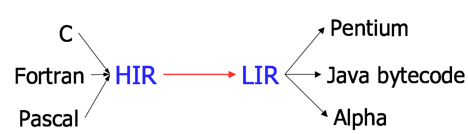
- Usually two IRs:

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 based on properties of source language (e.g. inlining)

Low-Level IR

- Low-level representation is essentially an abstract machine
- Has low-level constructs
  - Unstructured jumps, registers, memory locations
- 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:
    a = b OP c
  - Tree representation (Tiger Book)
  - Stack machine (like Java bytecode)
- Advantages:
  - Three-address code: easier dataflow analysis
  - Tree IR: easier instruction selection
  - Stack machine: easier to generate

Three-Address Code

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

```
a = (b+c)*(-e);        t1 = b + c
                       t2 = – e
                       a = t1 * t2
```


IR Instructions

- Assignment instructions:
  - a = b OP c : binary operation
    - arithmetic: ADD, SUB, MUL, DIV, MOD
    - logic: AND, OR, XOR
    - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - a = OP b : unary operation
    - Arithmetic MINUS or logic NEG
  - a = b : copy instruction
  - a = load b : load instruction
  - a = [b] : store instruction
  - [a] = b : symbolic address

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

- Models a load/store abstract machine

  a = [b]   [a] = b

- Take symbolic addresses of variables:

  a = addr b

Branch Instructions

- Unconditional jump: go to statement after label L

  jump L

- Conditional jump:
  - Test condition variable a
  - If value is true, jump to label L

  cjump 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
- 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