CSE 401 – Compilers

Lecture 12: A Survey of Intermediate Representations

Michael Ringenburg

Winter 2013

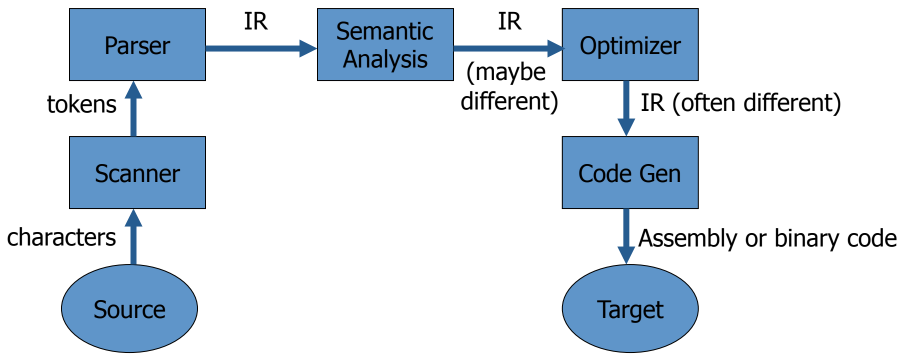
Reminders

- Homework 2 due by the end of the day today
- Project Part 2 due in 9 days
  - Part 2 typically takes a bit longer than part 1, so don't put it off

Agenda

- Survey of Intermediate Representations
  - Graphical Representations
    - Control Flow Graph
    - Dependence Graph
    - Concrete/Abstract Syntax Trees (ASTs)
  - Linear Representations
    - Stack based
    - 3 address
- We will go into some of these in more depth as they come up later in the course

Compiler Structure



Intermediate Representations

- In most compilers, the parser builds an intermediate representation of the program
  - Typically an AST, like your MiniJava compilers
- Rest of the compiler transforms the IR to improve ("optimize") it and eventually translates it to final code
  - Typically will transform initial IR to one or more lower level IRs along the way
- Some high-level examples today; more specific details as we cover later topics

IR Design Considerations

- Decisions affect speed and efficiency of the rest of the compiler
  - General rule: Compile time is important, but performance of executable *often* more important.
  - Typical use case is compile few times, run many times.
  - So make choices that improve compile time, as long as they don't impact performance of generated code.
  - "Coffee Break Rule": Compilation of a reasonable sized code shouldn't take longer than an average coffee break, or customers will complain.

IR Design

- Desirable properties
  - Easy to generate
  - Easy to manipulate
  - Expressive
  - Appropriate level of abstraction
- Different tradeoffs depending on compiler goals
- Different tradeoffs in different parts of the same compiler
  - Leads to different IRs in different parts.

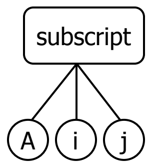
IR Design Dimensions

- Structure
  - Graphical (trees, graphs, etc.)
  - Linear (code for some abstract machine)
  - Hybrids are common (e.g., control-flow graphs with linear code in basic blocks)
- Abstraction Level
  - High-level, near to source language
  - Low-level, closer to machine, more exposed to compiler

Example: Array Reference

Source: A[i,j]

AST:



High level linear:

t1 ← A[i,j]

Low-level linear (3 address):

```
loadI 1     => r1
sub   rj,r1 => r2
loadI 10    => r3
mult  r2,r3 => r4
sub   ri,r1 => r5
add   r4,r5 => r6
loadI @A    => r7
add   r7,r6 => r8
load  r8    => r9
```

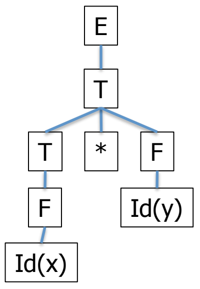
Levels of Abstraction

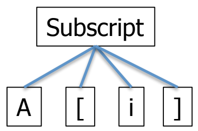
- Key design decision: how much detail to expose
  - Affects possibility and profitability of various optimizations
    - Depends on phase: Semantic Analysis, some optimizations prefer high level. Other optimizations, resource allocation, code generation prefer low level.
  - Most high-level IRs are graphical
    - But graphical IRs are also used with low-level
  - Linear IRs are typically low-level
  - But these generalizations *don't always hold*

Graphical IRs

- IRs represented as a graph (or tree)
- Nodes and edges typically reflect some structure of the program
  - E.g., source, control flow, data dependence
- May be large (especially syntax trees)
- High-level examples: Syntax trees, DAGs
  - Generally used in early phases of compilers
- Other examples: Control flow graphs and data dependence graphs
  - Often used in optimization and code generation

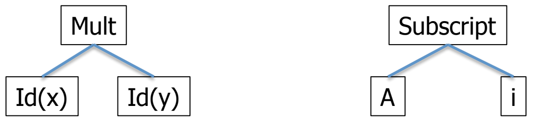
Graphical IR: Concrete Syntax Trees





- The full grammar is needed to guide the parser, but contains many extraneous details
  - E.g., syntactic tokens, rules that control precedence
- Typically the full syntax tree does not need to be used explicitly

Graphical IR: Abstract Syntax Trees

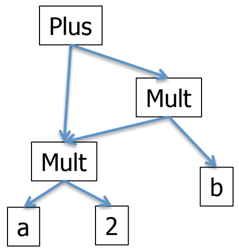


- Want only essential structural information (omit extra junk)
- Can be represented explicitly as a tree or in a linear form, e.g., in the order of a depth-first traversal. For a[i+j], this might be:

```
Subscript
  Id(A)
  Plus
    Id(i)
    Id(j)
```

- Common output from parser; used for static semantics (type checking, etc.) and sometimes high-level optimizations

Graphical IR: DAG



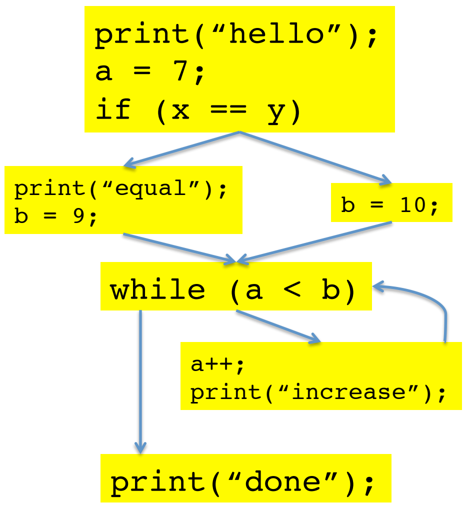
- DAG = Directed Acyclic Graph
  - In compilers, typically used to refer to an AST like structure, where common components may be reused. E.g, the 2*a in 2*a + 2*a*b (above).
  - Pros: Saves space, makes common subexpressions explicit.
  - Cons: If want to change just one occurence, need to split off. If variable value may change between evaluations, may not want to treat as common.

Control Flow Graph (CFG)

- Nodes are *Basic Blocks*
  - Code that always executes together (i.e., no branches into or out of the middle of the block).
  - I.e., “straightline code”
- Edges represent paths that control flow could take.
  - I.e., possible execution orderings.
  - Edge from Basic Block A to Basic Block B means Block B could execute immediately after Block A completes.
- Required for much of the analysis done in the optimizer.

CFG Example

```
print(“hello”);
a = 7;
if (x == y) {
  print(“equal”);
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print(“increase”);
}
print(“done”);
```



```
print(“hello”);
a = 7;
if (x == y)
```

```
print(“equal”);
b = 9;
```

```
b = 10;
```

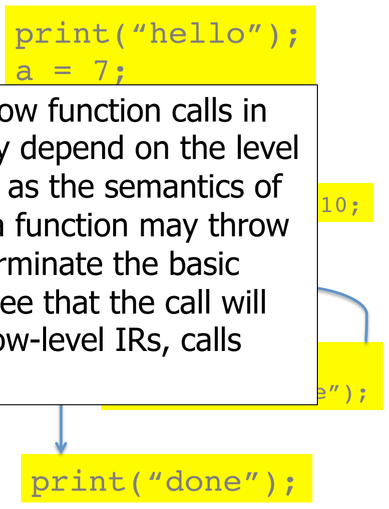
```
while (a < b)
```

```
a++;
print(“increase”);
```

```
print(“done”);
```

CFG Example

```
print("hello");
a = 7;
if
}
}
wh
print("increase");
}
print("done");
```



Note: There are variations in how function calls in basic blocks are treated. It may depend on the level of abstraction of the IR, as well as the semantics of the language. For example, if a function may throw an exception, the call should terminate the basic block (since there is no guarantee that the call will return to the same point). In low-level IRs, calls may also terminate blocks.

(Program/Data) Dependence Graph

- Often used in conjunction with another IR.
- In a data dependence graph, edges between nodes represent "dependencies" between the code represented by those nodes.
  - If A and B access the same data, and A must occur before B to achieve correct behavior, then there is a dependence edge from A to B.
  - A→B means compiler can't move B before A.
  - Granularity of nodes varies. Depends on abstraction level of rest of IR. E.g., nodes could be loads/stores, or whole statements.
  - E.g., a = 2; b = 2; c = a + 7;
    - Where's the dependence?

Types of dependencies

- Read-after-write (RAW)/“flow dependence”
  - E.g., a = 7; b = a + 1;
  - The read of ‘a’ must follow the write to ‘a’, otherwise it won’t see the correct value.
- Write-after-read (WAR)/“anti dependence”
  - E.g., b = a * 2; a = 5;
  - The write to ‘a’ must follow the read of ‘a’, otherwise the read won’t see the correct value.
- Write-after-write (WAW)/“output dependence”
  - E.g., a = 1; … a = 2; …
  - The writes to ‘a’ must happen in the correct order, otherwise ‘a’ will have the wrong final value.
- What about RAR/”input dependence”??

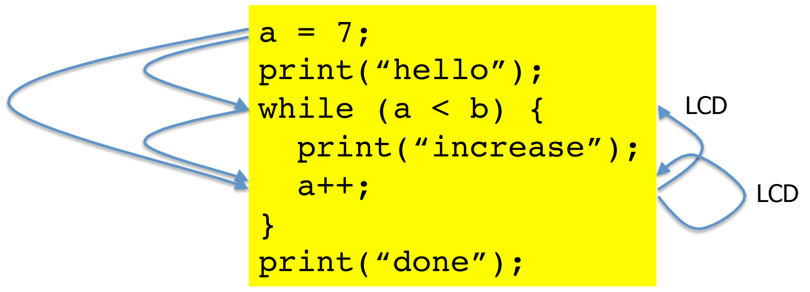
Loop-Carried Dependence

- *Loop carried dependence*: A dependence across iterations of a loop

```
for (i = 0; i < size; i++)
    x = foo(x);
```

- RAW loop carried dependence: the read of ‘x’ depends on the write of ‘x’ in the previous iteration
- Identifying and understanding these is critical for loop vectorization and automatic loop parallelization, because these may effectively reorder loop iterations.
  - If the compiler “understands” the nature of the dependence, it can sometimes be removed or dealt with
  - Often use sophisticated array subscript analysis for this

Dependence Graph Example



```
a = 7;
print(“hello”);
while (a < b) {
  print(“increase”);
  a++;
}
print(“done”);
```

LCD: Loop-Carried Dependence

Linear IRs

- Pseudo-code for some abstract machine
- Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked structures
- Examples: 3-address code, stack machine code

- Fairly compact
- Compiler can control reuse of names – clever choice can reveal optimizions.
- ILOC code

```
T1 ← 2
T2 ← b
T3 ← T1 * T2
T4 ← a
T5 ← T4 – T3
```

- Each instruction: pop operands, push result.
- Very compact
- Easy to create interpreter.
- Java bytecode

```
push 2
push b
multiply
push a
subtract
```

Abstraction Levels in Linear IR

- Linear IRs can also be close to the source language, very low-level, or somewhere in between.
- E.g., Linear IRs for C array reference a[i][j+2]
  - High-level: t1 ← a[i,j+2]

IRs for a[i,j+2], cont.

- Medium-level

```
t1 ← j + 2
t2 ← i * 20
t3 ← t1 + t2
t4 ← 4 * t3
t5 ← addr a
t6 ← t5 + t4
t7 ← *t6
```

- Low-level

```
r1 ← [fp-4]
r2 ←  r1 + 2
r3 ← [fp-8]
r4 ← r3 * 20
r5 ← r4 + r2
r6 ← 4 * r5
r7 ← fp – 216
f1 ← [r7+r6]
```

Abstraction Level Tradeoffs

- High-level: good for some high-level optimizations, semantic checking, but can't optimize things that are hidden (e.g., address calculations in subscript operations)
- Low-level: Needed for good code generation and resource utilization in back end, but lose some semantic knowledge (e.g., variables)
- Medium-level: Exposes more, but still keeps some semantic knowledge.
- Many compilers use all three at different phases.

Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- Most common example: control-flow graph
  - Nodes: basic blocks. Within nodes, linear representation of basic block's code.
- May also see Dependence Graph implemented as edges between linear instructions.
  - Possibly even inside CFG basic blocks

What IR to Use?

- Common choice: all(!)
  - AST or other structural representation built by parser and used in early stages of the compiler
    - Closer to source code
    - Good for semantic analysis
    - Facilitates some higher-level optimizations
  - Hybrid IR for optimization
  - Lower to low-level linear IR for later stages of compiler
    - Closer to machine code
    - Exposes machine-related optimizations
    - Good for resource allocation and scheduling

Coming Attractions

- Next 2-3 lectures – stuff you need for project part 3: semantic analysis, type checking, and symbol tables.
- Then, x86 overview (for code gen – project part 4).