COMP 181

Lecture 13
Intermediate representations and code generation
October 26, 2005

Midterm
- Scanners
  - Regular expressions, regular languages
  - Thompson's construction
  - Subset construction
  - Table representation
- Grammars
  - Context-free grammars
  - Sentence derivation, sentential forms
  - Rightmost, leftmost derivation
  - Precedence
  - Ambiguity

Midterm
- Top-down parsing
  - Left recursion
  - Predictive parsing
  - Lookahead
  - FIRST and FOLLOW sets
  - LL(1) property
  - Recursive descent
  - Left factoring
  - Table-driven predictive parsing

Midterm
- Bottom-up parsing
  - Handles
  - Reductions
  - Shift-reduce parsing
    - General skeleton
    - Shift-reduce, reduce-reduce conflicts
  - LR parser construction
    - LR items
    - Closure and goto functions
    - Canonical collection of items

Midterm
- Syntax-directed translation
  - Attribute grammars
    - Inherited, synthesized attributes
    - Evaluation
    - Ad-hoc SDT – integration into LR parsing
  - Static checking
    - Type systems
    - Type checking rules
    - Type equivalence
- Procedure abstraction
  - Nested scopes, symbol tables
  - Run-time vs compile-time
  - Storage layout, activation records
- Object-oriented languages
  - Prefixing – fields and methods
Prelude

- What is this?
  Hurricane
- Current storm: Wilma
  What’s next?
  Alpha, Beta, etc.
- What is “category X” storm?
  Saffir-Simpson Scale: wind speed and storm surge

Intermediate Representations

- Decisions in IR design affect the **speed** and **efficiency** of the compiler
- Some important IR properties
  - Ease of generation
  - Ease of manipulation
  - Size of the representation
  - Expressiveness
  - Level of abstraction
- Often, different IRs for different jobs

Types of IRs

- Three major categories
  - Structural
    - Graph oriented
    - Heavily used in source-to-source translators
    - Tend to be large
  - Linear
    - Pseudo-code for an abstract machine
    - Level of abstraction varies
    - Simple, compact data structures
    - Easier to rearrange
  - Hybrid
    - Combination of graphs and linear code

Level of Abstraction

- The level of **detail** exposed in an IR influences the profitability and feasibility of optimizations.
- Two different representations of an array reference:

  ```plaintext
  High level AST: Good for memory disambiguation
  
  Low level linear code: Good for address calculation
  ```
Abstract Syntax Tree
- AST: parse tree with some intermediate nodes removed
  - Example: \( x - 2 \times y \)
- What is this representation good for?
  - We can reconstruct original source
  - Source-to-source translators
  - Program understanding tools

Directed Acyclic Graph
- A directed acyclic graph (DAG)
  - AST with a unique node for each value
  - Example: \( z = x - 2 \times y \)
  - Why do this?
    - More compact (sharing)
    - Encodes redundancy

Stack Machine Code
- Originally for stack-based computers
  - Example: \( x - 2 \times y \)
- What are advantages?
  - Introduced names are implicit, not explicit
  - Simple to generate and execute code
  - Compact form – who cares about code size?
  - Systems where code is transmitted (the ‘Net)

Three Address Code
- Several variations of three address code
  - General form: \( x \leftarrow y \text{ op } z \)
  - With 1 operator (\text{ op }) and, at most, 3 names (x, y, z)
  - Example: \( z = x - 2 \times y \)
  - What are advantages?
    - Resembles many machines
    - Introduces a new set of names – we’ll need these later
    - Fairly compact form

Three Address Code: Quads
- Naïve representation of three address code
  - Table of \( k \times 4 \) small integers
  - Simple record structure
  - Easy to reorder
  - Explicit names
  - The original FORTRAN compiler used “quads”

Three Address Code: Triples
- Index used as implicit name
  - 25% less space consumed than quads
  - Much harder to reorder
  - Implicit names take no space!
Three Address Code: Indirect Triples

- List triples in a statement list data structure
- Implicit name space
- Uses more space than triples, but easier to reorder

<table>
<thead>
<tr>
<th>stmt array</th>
<th>load</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(101)</td>
<td>load</td>
<td>2</td>
</tr>
<tr>
<td>(102)</td>
<td>muli</td>
<td>(100)</td>
</tr>
<tr>
<td>(103)</td>
<td>load</td>
<td>x</td>
</tr>
<tr>
<td>(104)</td>
<td>sub</td>
<td>(103)</td>
</tr>
</tbody>
</table>

Control-flow Graph (CFG)

- Models the transfer of control in the procedure
- Nodes in the graph are *basic blocks*
- Within basic block: quads or any other linear form
- Edges in the graph represent control flow

Example:

```
if (x = y)
  a ← 2
  b ← 5
  a ← 3
  b ← 4
  a ← a + b
```

Static Single Assignment

- Idea:
  - Each variable assigned only once
  - A variable represents a specific value
  - Add on to other forms, mostly CFG

```
x = z + y
x = x + 1
y = x + 2
x = y - 1
```

- Turns imperative code into functional code

Static Single Assignment

- Advantages
  - Speeds up some program analysis
  - Functional semantics easier to reason about
  - Breaks up live ranges

- Disadvantages
  - Makes some optimizations more complex
  - Doesn’t work for C – why?
    - Pointers to local variables: p = &x; *p = 7;

Static Single Assignment

- Problem: what about control flow?

```
if (x = y)
  a ← 2
  b ← 5
  a ← 3
  b ← 4
  a ← a + b
```

- \( \Phi \) functions are selectors
- Works on loops as well

IR Memory Models

Two major models
- Register-to-register model
  - Keep all values that can be stored in a register in registers
  - Ignore machine limitations on number of registers
  - Compiler back-end must insert loads and stores

- Memory-to-memory model
  - Keep all values in memory
  - Only promote values to registers directly before use
  - Compiler back-end can remove loads and stores

- Compilers for RISC usually use register-to-register
The Rest of the Story…
Representing the code is only part of an IR.
There are other necessary components:
- Symbol table (already discussed)
- Constant table
  - Representation, type
  - Storage class, offset
- Storage map
  - Overall storage layout
  - Overlap information
  - Virtual register assignments

Multiple Representations
- Repeatedly lower the level of the IR
  - Each IR is suited to certain optimizations
  - Refine operations into next level
- Example:
  - High level: array operations: \( A[x] \)
  - Low level: address operations: \( p = A + x \times \text{sizeof()} \)

Code Generation
- Fast stuff followed by some very hard problems
  - The hard stuff: mostly code generation and optimization
  - For superscalars: allocation & scheduling that is particularly important

Structure of a Compiler
- We assume the following model
- Selection is fairly simple (problem of the 1980s)
- Allocation & scheduling are complex
- We’ll cover some analysis and optimization later

Definitions
- Instruction selection
  - Mapping IR into assembly code
  - Combining operations, using address modes
- Instruction scheduling
  - Reordering operations (Why?)
  - Changes demand for registers
- Register allocation
  - Deciding which values will reside in registers
  - Concerns about placement of memory operations

How hard are these problems?
- Instruction selection
  - Can make locally optimal choices, with automated tool
  - Global optimality is probably NP-Complete
  - Subset sum?
- Instruction scheduling
  - Single basic block ⇒ heuristics work quickly
  - General problem, with control flow ⇒ NP-Complete
- Register allocation
  - One basic block, no spilling, & 1 register size ⇒ linear time
  - Whole procedure is NP-Complete
  - Graph coloring
How hard are these problems?

- Recent research: **Denali**
  - Find optimal sequence of instructions
  - Uses iterative theorem proving technique:
    - Prove: "code cannot be implemented in X instructions"
    - If proof succeeds, increment X
    - If proof fails, counter-example shows instructions
  - Caveats:
    - Only works for X = 8 to 10 instructions!
    - No control flow
- Used by Google to optimize inner search loop

Next time…

- Monday, Oct 31 – midterm
- More code generation