



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

2

Midterm



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

3

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

4

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

5

Midterm

- Procedure abstraction
 - Nested scopes, symbol tables
 - Run-time vs compile-time
 - Storage layout, activation records
- Object-oriented languages
 - Prefixing – fields and methods

6

Prelude

- What is this?
Hurricane

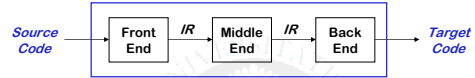


- Current storm: Wilma
What's next?
Alpha, Beta, etc.

- What is "category X" storm?
Saffir-Simpson Scale: wind speed and storm surge



Intermediate Representations



- Front end
 - Produces an intermediate representation (IR)
- Middle end
 - Transforms the IR to improve performance
- Back end
 - Transforms the IR into native code
- IR encodes the compiler's knowledge of the program
- Middle end usually consists of several passes



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

Examples: Trees, DAGs
- Linear
 - Pseudo-code for an abstract machine
 - Level of abstraction varies
 - Simple, compact data structures
 - Easier to rearrange

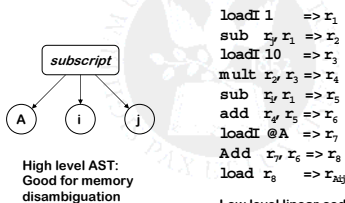
Examples: 3 address code, Stack machine code
- Hybrid
 - Combination of graphs and linear code

Example: Control-flow graph



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:



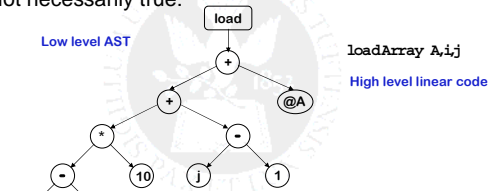
High level AST:
Good for memory disambiguation

Low level linear code:
Good for address calculation



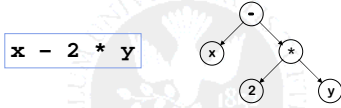
Level of Abstraction

- Structural IRs are usually considered high-level
- Linear IRs are usually considered low-level
- Not necessarily true:



Abstract Syntax Tree

- AST: parse tree with some intermediate nodes removed



- What is this representation good for?
 - We can reconstruct original source
 - Source-to-source translators
 - Program understanding tools

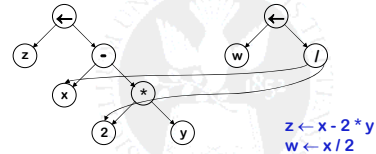


Tufts University Computer Science

13

Directed Acyclic Graph

- A directed acyclic graph (DAG)
 - AST with a unique node for each value



- Why do this?
 - More compact (sharing)
 - Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!

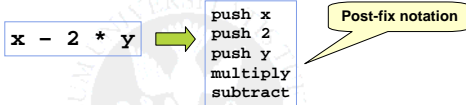


Tufts University Computer Science

14

Stack Machine Code

- Originally for stack-based computers



- What are advantages?
 - Introduced names are *implicit*, not *explicit*
 - Simple to generate and execute code
 - Compact form – who cares about code size?
 - Embedded systems
 - Systems where code is transmitted (the 'Net')



Tufts University Computer Science

15

Three Address Code

- Several variations of three address code

General form:

$$x \leftarrow y \text{ op } z$$

With 1 operator (op) and, at most, 3 names (x, y, z)

- Example: $z = x - 2 * y$ \rightarrow $t \leftarrow 2 * y$
 $z \leftarrow x - t$
- What are advantages?
 - Resembles many machines
 - Introduces a new set of names – we'll need these later
 - Fairly compact form



Tufts University Computer Science

16

Three Address Code: Quads

- Naïve representation of three address code
 - Table of $k * 4$ small integers
 - Simple record structure
 - Easy to reorder
 - Explicit names

The original FORTRAN compiler used "quads"

```
load r1, y
loadI r2, 2
mult r3, r2, r1
load r4, x
sub r5, r4, r3
```

RISC assembly code

load	1	Y	
loadi	2	2	
mult	3	2	1
load	4	X	
sub	5	4	2

Quadruples



Tufts University Computer Science

17

Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	load	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Implicit names take no space!



Tufts University Computer Science

18

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

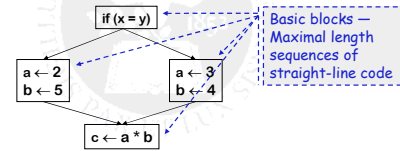
stmt array				
(100)	(100)	load	y	
(101)	(101)	loadl	2	
(102)	(102)	mult	(100)	(101)
(103)	(103)	load	x	
(104)	(104)	sub	(103)	(102)



Control-flow Graph (CFG)

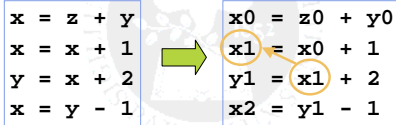
- Models the transfer of control in the procedure
 - Nodes in the graph are **basic blocks**
 - Within basic block: quads or any other linear form
 - Edges in the graph represent control flow

Example:



Static Single Assignment

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

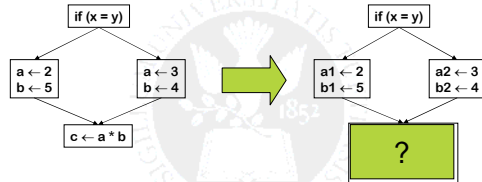


- Turns imperative code into functional code



Static Single Assignment

- Problem: what about control flow?



- Φ functions are selectors
- Works on loops as well



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;`



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

Often called **virtual registers**



The Rest of the Story...

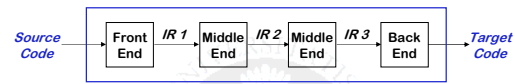
Representing the code is only part of an *IR*

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



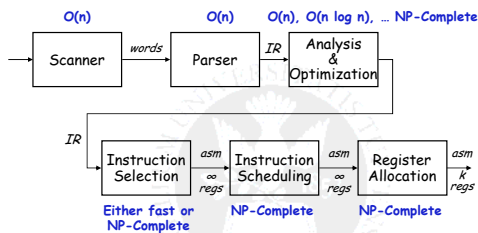
Multiple Representations



- 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: $p = A + x * \text{sizeof}()$



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

Superscalar processor:
we can hide latency

These 3 problems
are tightly coupled.



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 \Rightarrow heuristics work quickly
 - General problem, with control flow \Rightarrow NP-Complete
- Register allocation
 - One basic block, no spilling, & 1 register size \Rightarrow 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

