## CSE 501: Compiler Construction

Topic: implementing languages efficiently on stock hardware
- analysis & optimizations
- imperative, functional, and object-oriented languages

Prerequisites:
- CSE 401 or equivalent
- CSE 505 or equivalent

Read Appel's "Modern Compiler Implementation"
+ ~20 papers from literature

Periodic homework assignments + course project

Midterm + final



## Course outline

Basic compilation techniques:
- intermediate representations, analysis frameworks, optimizations

Fancier techniques:
- lattice-theoretic analysis frameworks & tools
- better intermediate representations

Call optimizations:
- inlining
- interprocedural analysis
  - with first-class functions
  - with dynamically dispatched messages
- procedure specialization & partial evaluation

Pointer analysis & optimizations

Dependence analysis for parallelization & loop restructuring

Register allocation

Instruction scheduling

Garbage collection algorithms

Compiling functional & object-oriented languages



## Goals/issues in language implementation

Correctness

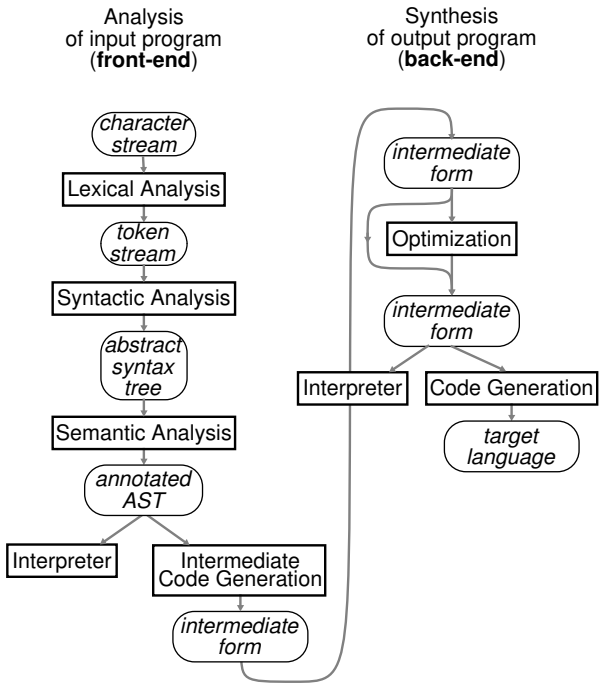
Efficiency
- what: time, data space, code space
- when: compile-time, run-time

Programming environment features
- fast turnaround
- separate compilation, shared libraries
- source-level debugging
- profiling
- garbage collection
- ...

Support expressive language features
- higher-order functions
- dynamic dispatching
- ...



## Standard compiler organization



## Key questions

How are programs represented in the compiler?

How are analyses organized/structured?
What analysis algorithms are used?
Over what region of the program are analyses performed?
When are optimizations performed?

What kinds of optimizations can be performed?
Which are profitable in practice?
How should analyses/optimizations be sequenced/combined?

What optimizations can be performed to handle:

- pointers, arrays
- parallel target machines
- first-class functions
- inheritance & message passing

Other issues:

- speeding compilation
- making compilers portable, table-driven
- supporting tools like debuggers, profilers, garbage collect'rs



## Overview of optimizations

First **analyze** program to learn things about it
Then **transform** the program based on info
Repeat...

Requirement: don't change the semantics!

- transform input program into
  semantically equivalent but better output program

Analysis determines when transformations are:

- legal
- profitable

Caveat: "optimize" a misnomer

- almost never optimal
- sometimes slow down some programs on some inputs
  (although hope to speed up most programs on most inputs)



## Semantics

Exactly what are the semantics that are to be preserved?
Subtleties:

- evaluation order
- arithmetic properties like associativity, commutativity
- behavior in "error" cases

Some languages very precise

- programmers always know what they're getting

Others weaker

- allow better performance (but how much?)

Semantics selected by compiler option?



## Scope of optimizations

**Peephole**: across a small number of "adjacent" instructions
[adjacent in space or time]

- trivial analysis

**Local**: within a **basic block**

- simple analysis

**Intraprocedural** (a.k.a. **global**):
across basic blocks, within a procedure

- analysis more complex:
  branches, merges, loops

**Interprocedural**:
across procedures, within a whole program

- analysis even more complex:
  calls, returns
- sometimes useful
  - more useful for higher-level languages
- hard with separate compilation

## Catalog of optimizations/transformations

arithmetic simplifications

- constant folding

```
x := 3 + 4   ⇒   x := 7
```

- strength reduction

```
x := y * 4   ⇒   x := y << 2
```

constant propagation

```
x := 5       ⇒  x := 5       ⇒  x := 5
y := x + 2      y := 5 + 2      y := 7
```

copy propagation

```
x := y       ⇒  x := y
w := w + x      w := w + y
```

common subexpression elimination (CSE)

```
x := a + b   ⇒  x := a + b
...             ...
y := a + b      y := x
```

partial redundancy elimination (PRE)

- CSE, but with earlier expression only available along subset of possible paths



integer range analysis

```
index := 5
if index >= 10 goto _error
a[index] := 0
```

pointer/alias analysis

```
p  := &x    ⇒    p  := &x      ⇒    p  := &x
*p := 5          *p := 5            *p := 5
y  := x + 1      y  := 5 + 1        y  := 6
```

dead (unused) assignment elimination

```
x := 5
... // no use of x
x := 6
```

dead (unreachable) code elimination

```
  if false goto _else
  ...
  goto _done
_else:
  ...
_done:
```



## Loop optimizations

loop-invariant code motion (a.k.a. code motion, code hoisting)

```
for j := 1 to 10      ⇒  for j := 1 to 10
  for i := 1 to 10           t := b[j]
    a[i] := a[i] + b[j]      for i := 1 to 10
                               a[i] := a[i] + t
```

induction variable elimination

```
for i := 1 to 10  ⇒  for p := &a[1] to &a[10]
  a[i] := a[i] + 1      *p := *p + 1
```

- `a[i]` is several instructions, `*p` is one

loop unrolling

```
for i := 1 to 3     ⇒  a[1] := a[1] + 1
  a[i] := a[i] + 1     a[2] := a[2] + 1
                       a[3] := a[3] + 1
```



parallelization

```
for i := 1 to 1000  ⇒  forall i := 1 to 1000
  a[i] := a[i] + 1        a[i] := a[i] + 1
```

loop interchange, skewing, reversal, ...

blocking

- restructuring loops for better data cache locality

## Call optimizations

inlining

```
l := ...        ⇒  l := ...     ⇒  l := ...
w := 4             w := 4          w := 4
a := area(l,w)     a := l * w      a := l << 2
```

interprocedural constant propagation, alias analysis, etc.

procedure specialization

- more generally, partial evaluation



## Machine-dependent optimizations

register allocation

instruction selection

```
p1 := p + 4    ⇒    ld %g3, [%g1 + 4]
x := *p1
```

instruction scheduling

```
ld  %g2, [%g1 + 0]  ⇒    ld  %g2, [%g1 + 0]
add %g3, %g2, 1          ld  %g5, [%g1 + 4]
ld  %g2, [%g1 + 4]       add %g3, %g2, 1
add %g4, %g2, 1          add %g4, %g5, 1
```



## The phase ordering problem

Typically, want to perform a number of optimizations;
in what order should the transformations be performed?

some optimizations create opportunities for other optimizations
⇒ order optimizations using this dependence

- some optimizations simplified
  if can assume another opt will run later & “clean up”

but what about cyclic dependencies?

- constant folding ⇔ constant propagation

what about adverse interactions?

- common subexpression elimination ⇔ register allocation
- register allocation ⇔ instruction scheduling



## Representation of programs

Primary goals:

- analysis is easy & effective
  - just a few cases to handle
  - provide support for linking things of interest
- transformations are easy
- general, across input languages & target machines

Additional goals:

- compact in memory
- easy to translate to and from
- tracks info for source-level debugging, profiling, etc.
- extensible (new optimizations, targets, language features)
- displayable

## High-level syntax-based representation

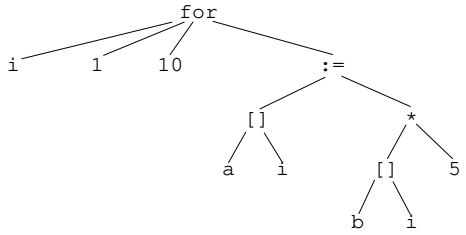
Represent source-level control structures & expressions directly

Examples
- (Attributed) AST
- Lisp S-expressions

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

AST:






## Low-level representation

Translate input programs into low-level primitive chunks, often close to the target machine

Examples
- assembly code, virtual machine code (e.g. stack machine)
- three address code, register transfer language (RTLs)
- lambda calculus

Standard RTL operators:

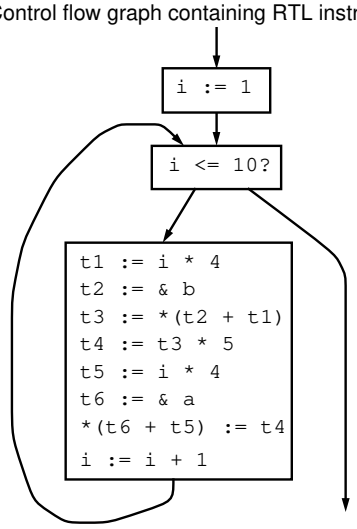
| | |
|---|---|
| assignment | `x := y;` |
| unary op | `x := op y;` |
| binary op | `x := y op z;` |
| address-of | `p := &y;` |
| load | `x := *(p + o);` |
| store | `*(p + o) := x;` |
| call | `x := f(...);` |
| unary compare | `op x ?` |
| binary compare | `x op y ?` |



Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

Control flow graph containing RTL instructions:






## Comparison

Advantages of high-level rep:
- analysis can exploit high-level knowledge of constructs
- easy to map to source code terms for debugging, profiling
- may be more compact
- supports semantics-based reasoning about correctness etc. of analysis

Advantages of low-level rep:
- can do low-level, machine-specific optimizations (if target-based representation)
  - high-level rep may not be able to express some transformations
- supports good analysis (?), transformations
- can have relatively few kinds of instructions to analyze
- can be language-independent
  - can be used e.g. at link-time

High-level rep suitable for a source-to-source optimizer

Can mix multiple representations in single compiler, can sequence compilers using different reps

## Components of representation

Operations

Control flow: sequencing of operations
- evaluation of then & else arms depends on result of test
- side-effects of statements occur in right order

Data flow: flow of values from **definitions** to **uses**
- operands computed before operation

Also called control & data **dependence**

Ideal: represent just those dependences that matter
- dependences constrain transformations
- fewest dependences ⇒ most flexibility in implementation



## Representing control dependences

Option 1: high-level representation
- control flow implicit in semantics of AST nodes

Option 2: control flow graph
- nodes are basic blocks
  - instructions in basic block sequence side-effects
- edges represent branches
  (control flow between basic blocks)

Fancier:
- control dependence tree,
  part of **program dependence graph** (PDG)
- convert into data dependences on explicit state,
  in **value dependence graph** [Weise *et al. 94*]
- ...



## Kinds of data dependences

read-after-write (RAW): **true dependence**, **flow dependence**
- reflects real data flow, operands to operation

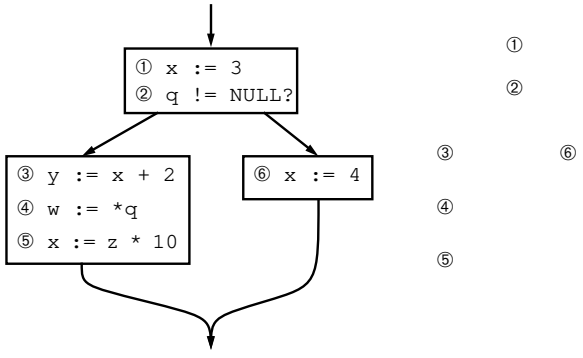
write-after-read (WAR): **anti-dependence**
write-after-write (WAW): **output dependence**
- reflects reuse of memory, not real data flow
  ⇒ can sometimes be eliminated by optimization

read-after-read (RAR): no dependence
- can occur in any order



## Example

```
x := 3
if q != NULL then
  y := x + 2
  w := *q
  x := z * 10
else
  x := 4
endif
```



## Representing data dependences

Within basic block:

Option 1: sequence of instructions
(represent data flow as a kind of control dependence)
- \+ simple, source-like
- \+ fixed ordering supports easy analysis
- – may overconstrain order of operations

Option 2: expression tree/DAG
- \+ natural, abstract
- \+ directly captures data dependences within basic block
- \+ DAG supports local CSE
- \+ can be compact
- – conceptually harder to analyze, transform
- – must linearize eventually



## Example

Source:

```
x := (z/y) + (y*4)
y := x + (y*4)
z := z + (y*4)
```

Linear RTL:

```
t1 := z/y
t2 := y*4
x := t1 + t2

t3 := y*4
y := x + t3

t4 := y*4
z := z + t4
```



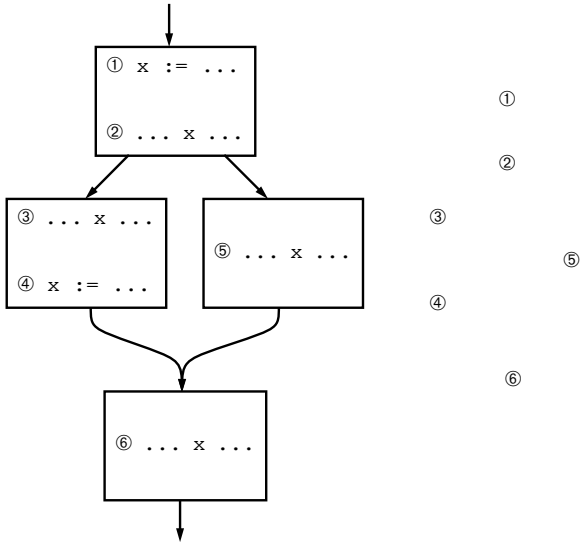
## Representing data dependences, cont.

Across basic blocks:

Option 1: implicitly through variable defs/uses
- \+ simple
- – analysis wants important things explicit

Option 2: def/use chains, linking each def with each use
- \+ explicit
- – must be computed, maintained after transformations
- – space consuming

Fancier def/use representations:
- **static single assignment** (SSA) form
- **dependence flow graphs** (DFGs)
- value dependence graphs
- ...



## Example