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 “Dragon Book” + ~20 papers from literature

Periodic homework assignments + course project

Midterm + final

Course project

A choice:
- a research & paper-based project
- an implementation-oriented project
  - sign up for 590-CP for 2 additional credits

Paper-based project

Task: identify a topic related to course, research it, write a ~15p. report on it, maybe orally present
- two-person teams

E.g.
- investigate, design, & specify some new optimization
- study opts for other kinds of languages (e.g. logic-based, constraint-based, parallel languages)
- study support for incremental compilation, source-level debugging of optimized code, profiling, etc.

Implementation project

Task: design, implement, and assess an optimization
- analysis & transformation components of optimization
- choose own project, negotiated with me

In teams of 2-4 people

Use some existing optimizing compiler infrastructure
Recommended: Vortex compiler
- \+ very OO, designed to be extensible w/ new optimizations
- \+ written in Cecil
- – written in Cecil
- – not as well documented or modular as desirable

Alternatives: ZPL compiler, SUIF, Multiflow, GCC...

Written group report
- project proposal due February 3
- specification of optimization draft due February 19
- full draft due March 3
- final paper due March 14

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
- procedure specialization & partial evaluation

Pointer analysis

Dependence analysis for parallelization & loop restructuring

Register allocation

Instruction scheduling

Garbage collection algorithms

Compiling functional languages

Compiling object-oriented languages

Goals/issues in language implementation

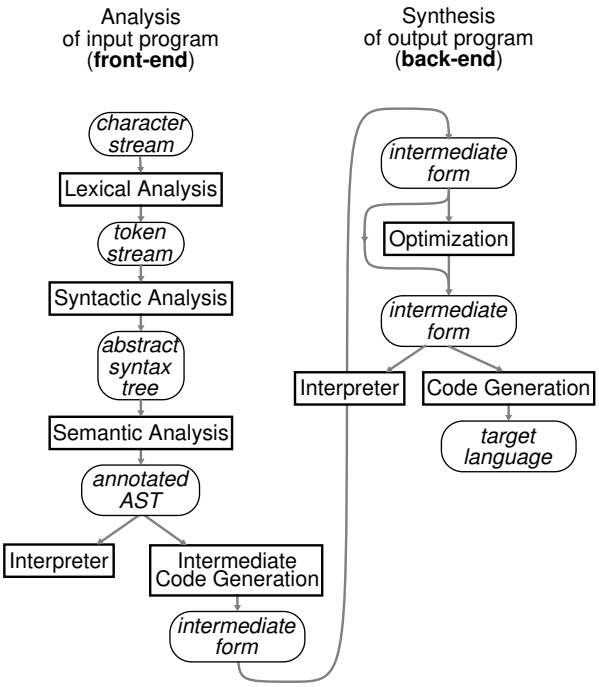
Correctness

Efficiency
- time & data space & code space
- 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?

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
- garbage collection

Other issues:
- speeding compilation
- making compilers portable, table-driven
- supporting tools like debuggers, profilers

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 couple of adjacent instructions

- trivial analysis
- surprisingly useful

Local: within a **basic block**

- simple analysis
- moderately effective

Global, **intraprocedural**:
across basic blocks, within a procedure

- analysis more complex:
  branches, merges, loops
- many useful optimizations

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
y := x + 2       y := 5 + 2   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)

integer range analysis

```
index := 5
if index >= 10 goto _error
```

pointer/alias analysis

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

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

```
for j := 1 to N      ⇒  for i' = 1 to N by K
  t := b[j]                 for j := 1 to N
  for i := 1 to N             t := b[j]
    a[i] := a[i] + t          for i := i' to i'+K
                                a[i] := a[i] + t
```


Call optimizations

inlining

```
len := ...           ⇒  len := ...
wid := 4                wid := 4
a := area(len, wid)     a := len * wid   len << 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