CSE 501: Compiler Construction

Topic: implementing languages efficiently on stock hardware

- optimizations, code generation, and run-time systems
- programming environment tools
- imperative, functional, and object-oriented languages

Prerequisites:

- CSE 401 or equivalent
- CSE 505 or equivalent

Lots of reading of papers from literature

Course project

Midterm + final

Homework?

Course project

Task: design, implement, and assess an optimization

- analysis & transformation components of optimization
- choose own project, negotiated with me

In teams of 2-5 people

Written group report

- proposal due January 30
- draft due February 27
- final version due March 8

Project infrastructure

Use Vortex compiler infrastructure for project

+ an existing optimizing compiler that works
+ includes iterative analysis framework, ...
+ can focus on your optimization, not infrastructure building

Vortex written in Cecil object-oriented language

+ more extensible
– need to learn Cecil
+ get to learn Cecil

Vortex compiles Cecil today

– fewer benchmarks
– harder to do some kinds of analyses, projects
+ some cool kinds of projects available

Vortex is a research project, under construction

– may have bugs, ugly code
+ not too big yet
+ you can have an impact!

Course outline

Week	Topic
1/2	***Holiday*** Structure of compilers; standard optimizations Intermediate representations: ASTs, CFGs, BBs, DAGs, RTLs Iterative global data flow analysis Common optimizations
1/9	Lattice-theoretic data flow analysis framework Data flow analysis generators
1/16	***Holiday*** Advanced intermediate representations: def/use chains, SSA form, PDGs
1/23	Inlining Interprocedural analysis Partial evaluation
1/30	Alias and pointer analysis Dependence analysis; automatic parallelization
MIDTERM over the weekend	
2/6	Global register allocation
2/13	Calling conventions; callee-save vs. caller-save Link-time register allocation Instruction scheduling
2/20	***Holiday*** Profiling Source-level debugging
2/27	Garbage collection Implementing functional languages
3/6	Implementing object-oriented languages

Goals/issues in language implementation

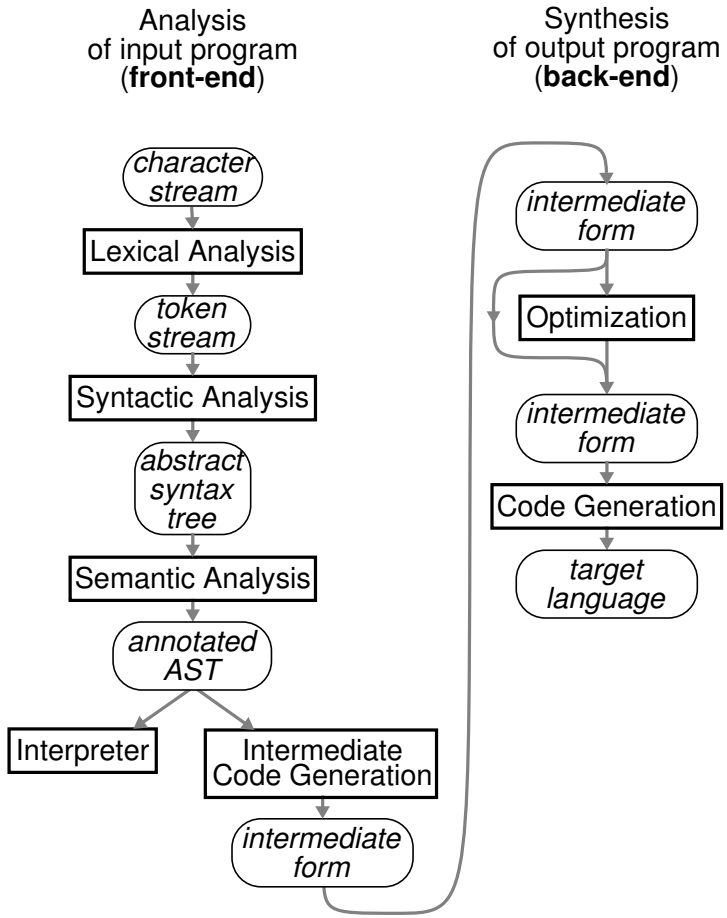
Correctness

Efficiency
- time & space
- compile-time & run-time

Programming environment features
- fast turnaround
- source-level debugging
- profiling
- ...

Support expressive language features
- higher-order functions
- message passing
- automatic GC
- ...

Conventional compiler organization



Overview of optimizations

First analyze program to learn things about it

Then make transformations to program based on info

Repeat...

Requirement: don't change the semantics!

- transform input program into
  semantically equivalent but
  more efficient output program

Themes:

- don't compute it if you don't have to
- compute it at compile-time
- use as fast a method of computing it as you can
  - optimize for common case
  - compute in parallel
- don't recompute it

Caveat: "optimize" a misnomer

Semantics

Exactly what are the semantics that are to be preserved?

E.g.:

- evaluation order
- behavior in "error" cases

Some languages very precise

- programmers always know what they're getting

Others weaker

- allow better performance (but how much?)

Selected by compiler option?

Classification of optimizations

Peephole: across a couple of 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
- 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
```

blocking

Call optimizations

inlining / procedure integration / unfolding / β-substitution

```
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

in what order should the transformations be performed?

some optimizations create opportunities for other optimizations
$\Rightarrow$ order optimizations using this dependence

but what about cyclic dependencies?

- constant folding $\Leftrightarrow$ constant propagation

what about adverse interactions?

- common subexpression elimination $\Leftrightarrow$ register allocation
- register allocation $\Leftrightarrow$ instruction scheduling