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

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

In teams of 2-4 people

Written group report
- project proposal due February 5
- draft due February 26
- final version due March 4

Project infrastructure

Use Vortex compiler infrastructure for project
- \+ an existing optimizing compiler that works
- \+ includes iterative dataflow 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, C/C++, Modula-3, etc. today
- – more complicated intermediate representation
- \+ can focus on "traditional" operators & input languages
- \+ some cool kinds of projects available

Vortex is a research project, under construction
- – may have bugs, ugly code
- \+ some cool features not found in regular compilers
- \+ springboard for future research/quals projects

Course outline

Week	Topic
1/1	***Holiday*** Standard intermediate representations, standard analysis frameworks, standard optimizations
1/8	Fancier analysis frameworks Analysis generators
1/15	***Holiday*** Fancier intermediate representations
1/22	Inlining Interprocedural analysis Procedure specialization (partial evaluation)
1/29	Alias and pointer analysis Dependence analysis; automatic parallelization
2/5	Global register allocation
2/13	Link-time register allocation Instruction scheduling
2/20	***Holiday*** Garbage collection
2/27	Implementing functional languages
3/6	Implementing object-oriented languages

Goals/issues in language implementation

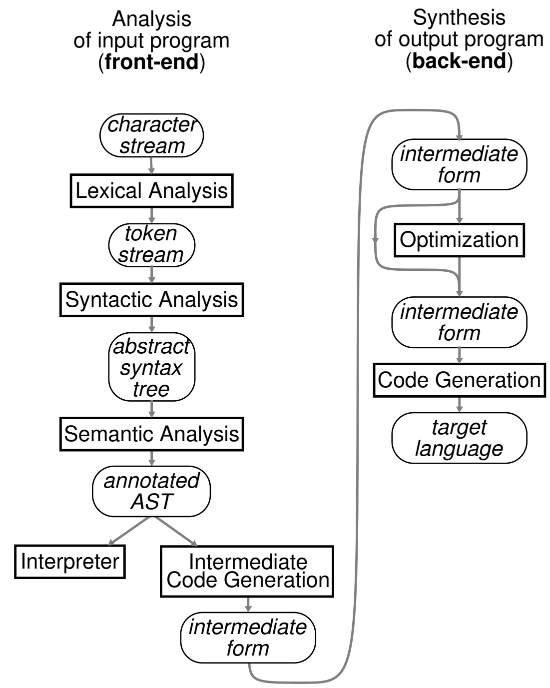
Correctness

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

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

Support expressive language features
- higher-order functions
- immutable data structures
- message passing
- ...

Conventional 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

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 if you can
- 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?

Scope 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, 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

Representation of programs

Primary goals:

- analysis is easy & effective
- transformations are easy
- just a few cases to handle
- general, across input languages & target machines

Additional goals:

- compact in memory
- easy to generate 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

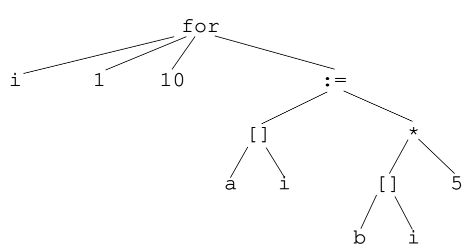
Examples

- (Attributed) AST
- Lisp S-expressions

Source:

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

AST:



Pros/cons

+ analysis exploits high-level knowledge of constructs
+ easy to map to source code terms for debugging, profiling
+ supports semantics-based reasoning about analysis, correctness, etc.

– some transformations hard to express
– low-level optimizations hard/impossible
– can be many kinds of AST nodes to analyze
– limited to particular input language(s)

Suitable for a source-to-source optimizer, parallelizer

- rely on another compiler to do low-level optimizations

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 (rtl's)
- lambda calculus

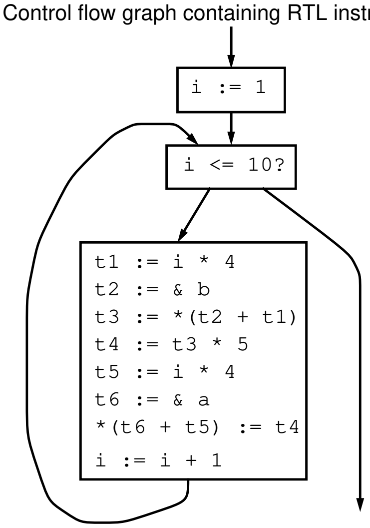
Standard RTL operators:

```
x := y;
x := op y;
x := y op z;

p := &y;
x := *(p + o);
*(p + o) := x;

x := f(...);

op x ?
x op y ?
```


Source:

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

Control flow graph containing RTL instructions:



Pros/cons

+ can do low-level, machine-specific optimizations
  (if target-based representation)
+ supports good analysis (?), transformations
+ can have relatively few kinds of instructions to analyze
+ can be language-independent

– may have to try to recover high-level knowledge
– hard to track source for debugging, profiling, etc.
– can be more space-, time-consuming

Idea: use multiple representations at different levels