CSE 501:
Implementation of Programming Languages

Main focus: **program analysis and transformation**
- how to represent programs?
- how to analyze programs? what analyses to perform?
- how to transform programs? what transformations to apply?

Applications to compilers and software engineering tools

Applied to imperative, functional, and object-oriented languages

Advanced language runtime systems

Readings:
- ~12 papers from literature
- Suggested reference books:
  - Cooper & Torczon's "Engineering a Compiler"
  - Appel's "Modern Compiler Implementation"
  - "Compilers: Principles, Techniques, & Tools" a.k.a. Dragon Book

Coursework:
- periodic homework assignments (~2-4)
- course project assignments (~2-3)
- midterm(?), final

Course outline

Models of compilation/analysis

Tour of standard optimizing transformations

Basic program representations and analyses
Fancier program representations and analyses

Interprocedural representations, analyses, and transformations
- for imperative, functional, and OO languages

Run-time system issues
- garbage collection
- compiling dynamic dispatch, first-class functions, ...

Dynamic (JIT) compilation

Why study compilers?

Meeting area of programming languages, architectures
- capabilities of compilers greatly influence their design

Program representation, analysis, and transformation is widely useful beyond pure compilation
- software engineering tools
- DB query optimizers, programmable graphics renderers (domain-specific languages and optimizers)
- safety/security checking of code, e.g. in programmable/extensible systems, networks, databases

Increasing applicability of other domains to compilers
- AI techniques to guide optimizers through search space

Cool theoretical aspects, too
- lattice domains, graph algorithms, computability/complexity

---

Goals for compilers

Be correct

Be efficient
- of: time, data space, code space
- at: compile-time, run-time

Support expressive, safe language features
- OO method dispatching
- first-class functions
- bounds-checked arrays, exceptions, continuations
- garbage collection
- reflection, dynamic code loading
- ...

Support desirable programming environment features
- fast turnaround
- separate compilation, shared libraries
- source-level debugging
- ...

Be implementable, maintainable, evolvable, ...

Key questions

How are programs represented in the compiler?

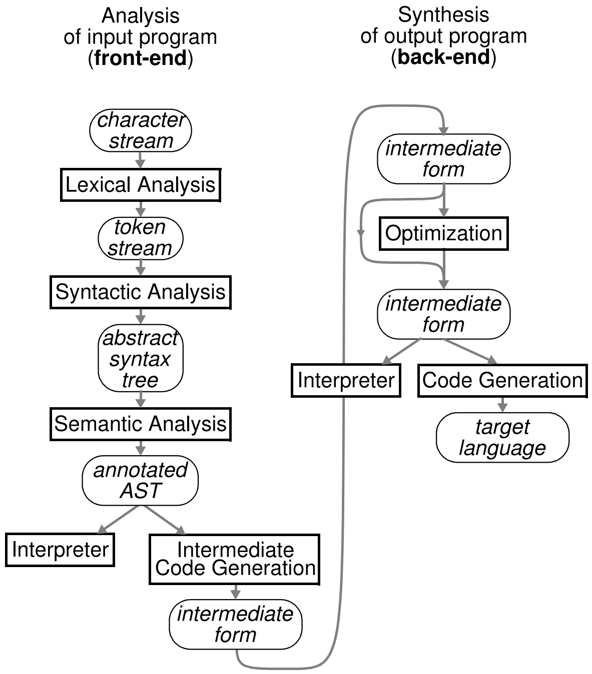
How are analyses organized/structured?
- Over what region of the program are analyses performed?
- What analysis algorithms are used?

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

How best to compile in face of:
- pointers, arrays
- first-class functions
- inheritance & message passing
- parallel target machines

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

Standard compiler organization



Compilation models

Separate compilation
- compile source files independently
- trivial link, load, run stages

+ quick recompilation after program changes

– poor interprocedural optimization

Link-time compilation
- delay (bulk of) compilation until link-time

+ allow interprocedural & whole-program optimizations

– quick recompilation?

– shared precompiled libraries?

– dynamic loading?

Examples: Vortex, Whirlwind (now),
some other research optimizers/parallelizers, ...

Run-time compilation (a.k.a. dynamic, just-in-time compilation)
- delay (bulk of) compilation until run-time
- can perform whole-program optimizations
- can perform opts based on run-time program state, execution environment

+ best optimization potential

+ can handle run-time changes/extensions to the program

– severe pressure to limit run-time compilation overhead

Examples: Java/.NET JITs, Dynamo, FX-32, Transmeta

Selective run-time compilation
- choose what part of compilation to delay till run-time

+ can balance compile-time/benefit trade-offs

Example: DyC

Hybrids of all the above
- spread compilation arbitrarily across stages

+ all the advantages, and none of the disadvantages!!

Example: Whirlwind (future)

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

- result is 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 analysis

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

- trivial analysis

Local: within a **basic block**

- simple, fast 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
- hard with separate compilation

Whole-program:
analysis can make closed-world assumptions

A tour of common 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
```

integer range analysis

- fold comparisons based on range analysis
- eliminate unreachable code

```
for(index = 0; index < 10; index ++) {
  ~~if index >= 10 goto _error~~
  a[index] := 0
}
```

- more generally, symbolic assertion analysis

common subexpression elimination (CSE)

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

- can also eliminate redundant memory references, branch tests

partial redundancy elimination (PRE)

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

```
if ... then ⇒  if ... then
  ...            ...
  x := a + b     t := a + b; x := t
end            else t := a + b end
...            ...
y := a + b     y := t
```

copy propagation

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

dead (unused) assignment elimination

```
~~x := y ** z~~
... // no use of x
x := 6
```

- a common clean-up after other optimizations:

```
x := y      ⇒  x := y      ⇒  ~~x := y~~
w := w + x     w := w + y  ⇒  w := w + y
... // no use of x
```

partial dead assignment elimination

- like DAE, except assignment only used on some later paths

dead (unreachable) code elimination

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

- another common clean-up after other optimizations

pointer/alias analysis

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

x  := 5
*p := 3
y  := x + 1 ⇒   ???
```

- augments lots of other optimizations/analyses

loop-invariant code motion

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

induction variable elimination

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

- `a[i]` is several instructions, `*p` is one
  - a kind of strength reduction

loop unrolling

```
for i := 1 to N      ⇒  for i := 1 to N by 4
  a[i+1] := a[i] + 1      a[i+1] := a[i]   + 1
                          a[i+2] := a[i+1] + 1
                          a[i+3] := a[i+2] + 1
                          a[i+4] := a[i+3] + 1
```

loop peeling, ...

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/tiling: restructuring loops for better cache locality

```
for i := 1 to 1000
  for j := 1 to 1000
    for k := 1 to 1000
      c[i,j] += a[i,k] * b[k,j]
⇒
for i := 1 to 1000 by TILESIZE
  for j := 1 to 1000 by TILESIZE
    for k := 1 to 1000
      for i' := i to i+TILESIZE
        for j' := j to j+TILESIZE
          c[i',j'] += a[i',k] * b[k,j']
```

inlining

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

- lots of “silly” optimizations become important after inlining

static binding of dynamic calls

- in imperative languages, for call of a function pointer:
  if can compute unique target of pointer,
  can replace with direct call
- in functional languages, for call of a computed function:
  if can compute unique value of function expression,
  can replace with direct call
- in OO languages, for dynamically dispatched message:
  if can deduce class of receiver,
  can replace with direct call
- other possible optimizations even if several possible callees

procedure specialization

register allocation

instruction selection

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

- particularly important on CISCs

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
```

- particularly important for instructions with delayed results, and on wide-issue machines
- less important on dynamically scheduled machines

Optimization themes

Don’t compute it if you don’t have to

- dead assignment elimination

Compute it at compile-time if you can

- constant folding, loop unrolling, inlining

Compute it as few times as possible

- CSE, PRE, PDE, loop-invariant code motion

Compute it as cheaply as possible

- strength reduction, induction var. elimination, parallelization, register allocation, scheduling

Enable other optimizations

- constant & copy propagation, pointer analysis

Compute it with as little code space as possible

- dead code elimination

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 dependences?

- e.g. constant folding ⇔ constant propagation

what about adverse interactions?

- e.g.
  common subexpression elimination ⇔ register allocation
- e.g.
  register allocation ⇔ instruction scheduling

Engineering

Building a compiler is an engineering activity

- balance
  complexity of implementation,
  speed-up of “typical” programs,
  compilation speed,
  ...

Near infinite number of speial cases for optimization
can be identified

- can’t implement them all

Good compiler design, like good language design, seeks
small set of powerful, general analyses and transformations,
to minimize implementation complexity while
maximizing effectiveness

- reality isn’t always this pure...

Representation of programs

Primary goals:

- analysis is easy & effective
  - just a few cases to handle
  - directly link related things
- transformations are easy to perform
- transformed programs are easy to express
- 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

Option 1: high-level syntax-based representation

Represent source-level control structures & expressions directly

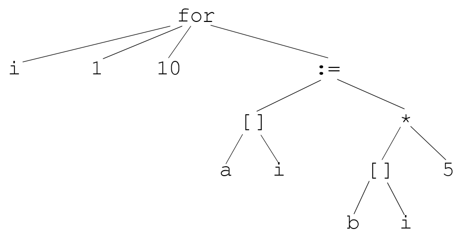
Examples

- (Attributed) AST
- Lisp S-expressions
- extended lambda calculus

Source:

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

AST:



Option 2: 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)

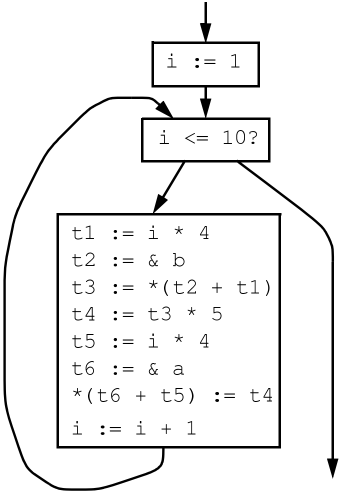
Standard RTL operators:

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:



```
i := 1
```

```
i <= 10?
```

```
t1 := i * 4
t2 := & b
t3 := *(t2 + t1)
t4 := t3 * 5
t5 := i * 4
t6 := & a
*(t6 + t5) := t4
i := i + 1
```

Comparison

Advantages of high-level rep:
- analysis can exploit high-level knowledge of constructs
  - probably faster to analyze
- easy to map to source code terms for debugging, profiling
- (may be) more compact

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
- can have relatively few kinds of instructions to analyze
- can be language-independent

High-level rep suitable for a source-to-source or special-purpose optimizer, e.g. inliner, parallelizer

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

Q: what about Java bytecodes?

Components of representation

Operations

Dependences between operations
- **control** dependences: sequencing of operations
  - evaluation of then & else arms depends on result of test
  - side-effects of statements occur in right order
- **data** dependences: flow of values from **definitions** to **uses**
  - operands computed before operation

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** (CFG)

- nodes are **basic blocks**
  - instructions in basic block sequence side-effects
- edges represent branches
  (control flow between basic blocks)

Option 2b: CFG whose nodes are individual instructions

Some fancier options:

- **control dependence graph**,
  part of **program dependence graph** (PDG)
  [Ferrante *et al.* 87]
- convert into data dependences on a memory state,
  in **value dependence graph** (VDG) [Weise *et al. 94*]

Representing data dependences

Option 1: **implicitly** through variable defs/uses in CFG

+ simple, source-like
– may overconstrain order of operations
– analysis wants important things explicit ⇒
  analysis can be slow

Option 2: **def/use chains**, linking each def to each use

- a kind of **data flow graph** (DFG)
+ explicit ⇒ analysis can be fast
– must be computed, maintained after transformations
– may be space-consuming

Some fancier options:

- **static single assignment** (SSA) form [Alpern *et al.* 88]
- value dependence graphs (VDGs)
- ...

Example