# CIS 341: Compilers

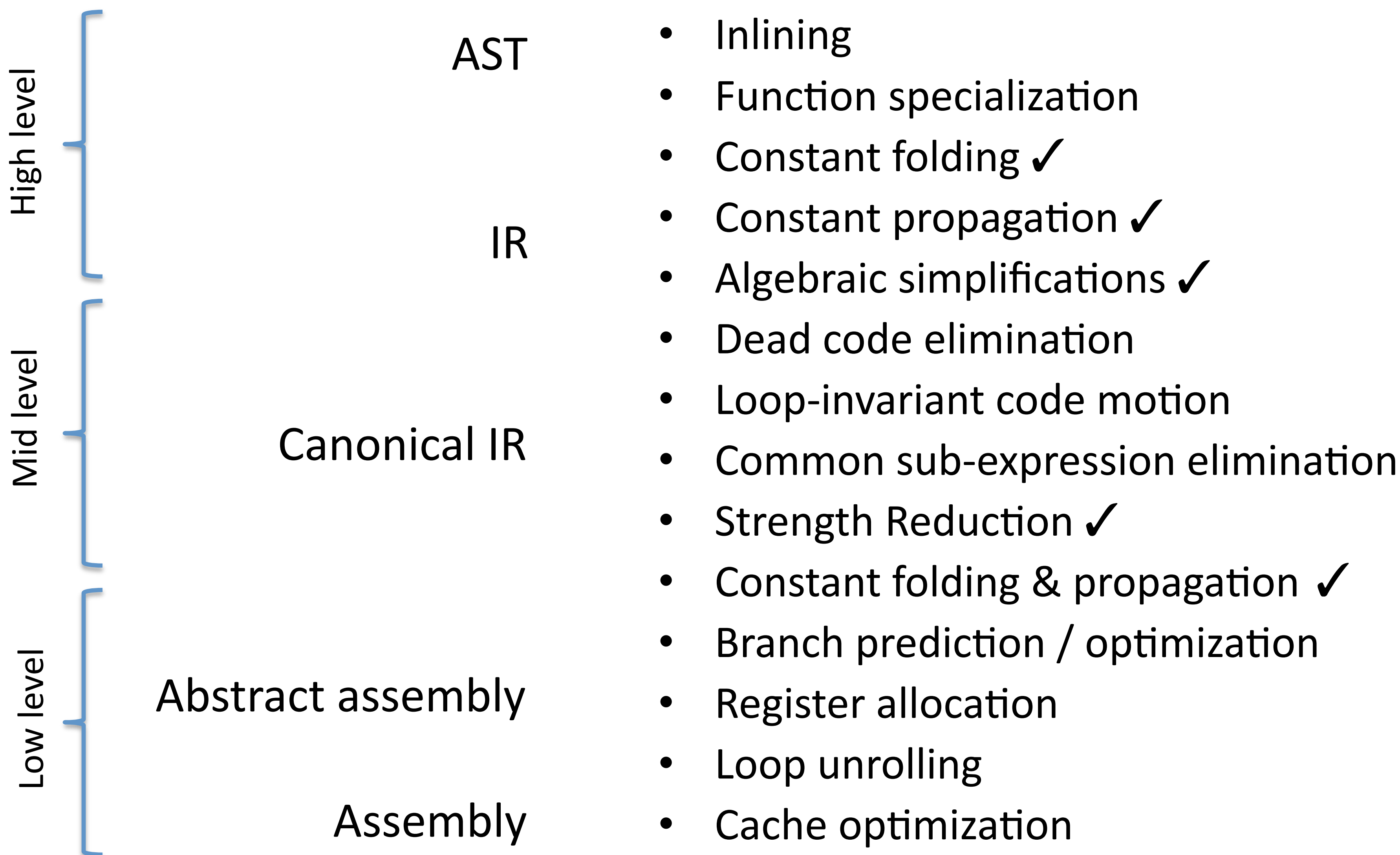
Lecture 26

# The Plan

- Today:
  - Continue with describing optimizations at a high level
  - Start developing the algorithms/tools for implementing optimizations

- Project 5: Full OAT compiler: objects and extended typechecking
  - Due: This Friday, November 7th at 11:59 pm

# A Variety of optimizations



# Unreachable/Dead Code

- Basic blocks not reachable by any trace leading from the starting basic block are *unreachable* and can be deleted.
  - Performed at the canonical IR or assembly level
  - Improves cache, TLB performance
- Dead code: similar to unreachable blocks.
  - A value might be computed but never subsequently used.
- Code for computing the value can be dropped
- But only if it's *pure*,i.e. it has *no externally visible side effects*
  - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
  - Note: Pure functional languages (e.g. Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

# Inlining

- Replace a call to a function with the body of the function itself with arguments rewritten to be local variables:
- Example in OAT code:

```
int g(int x) ( 1 + pow(x) )
int pow(int a) ( int b = 1; int n = 0;
   while (n < a) (b = 2 * b); b )
```

```
int g(int x) ( 1 + (int a = x; int b = 1; int n =0;
   while (n < a) (b = 2 * b); b) )
```

- May need to rename variable names to avoid *name capture*
  - Example of what can go wrong? Hint: suppose there is a global variable called x
- Best done at the AST or relatively high-level IR.
- When is it profitable?
  - Eliminates the stack manipulation, jump, etc.
  - Can increase code size.

# Code Specialization

- Idea: create specialized versions of a function that is called from different places with different arguments.
- Example: specialize function **`f`** in:

```
class A implements I { int m() {…} }
class B implements I { int m() {…} }
int f(I x) { x.m(); }       // don't know which m
A a = new A(); f(a);        // know it's A.m
B b = new B(); f(b);        // know it's B.m
```

- **`f_A`** would have code specialized to dispatch to **`A.m`**
- **`f_B`** would have code specialized to dispatch to **`B.m`**
- You can also inline methods when the run-time type is known statically
  - Often just one class implements a method.

# Common Subexpression Elimination

- In some sense it's the opposite of inlining: fold redundant computations together
- Example:

`a[i] = a[i] + 1` compiles to:

```
[a + i*4] = [a + i*4] + 1
```

Common subexpression elimination removes the redundant add and multiply:

```
t = a + i*4; [t] = [t] + 1
```

- For safety, you must be sure that the shared expression always has the same value in both places!

# Unsafe Common Subexpression Elimination

- Example: consider this OAT function:

```
unit f(int[] a, int[] b, int[] c) (
  int j = …; int i = …; int k = …;
  b[j] = a[i] + 1; c[k] = a[i]
)
```

- The following optimization that shares the expression **a[i]** is unsafe… why?

```
unit f(int[] a, int[] b, int[] c) (
  int j = …; int i = …; int k = …;
  t = a[i];
  b[j] = t + 1; c[k] = t
)
```

# Loop Optimizations

- Program hot spots often occur in loops.
  - Especially inner loops
  - Not always: consider operating systems code or compilers vs. a computer game or word processor
- Most program execution time occurs in loops.
  - The 90/10 rule of thumb holds here too
- Loop optimizations are very important, effective, and numerous
  - Also, concentrating effort to improve loop body code is usually a win

# Loop Invariant Code Motion (revisited)

- Another form of redundancy elimination.
- If the result of a statement or expression does not change during the loop *and* it's pure, it can be hoisted outside the loop body.
- Often useful for array element addressing code
  - Invariant code not visible at the source level

```
for (i = 0; i < a.length; i++) {
    /* a not modified in the body */
}
```

```
t = a.length;
for (i =0; i < t; i++) {
  /* same body as above */
}
```

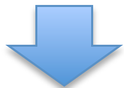
Hoisted loop-invariant expression

# Strength Reduction (revisited)

- Strength reduction can work for loops too
- Idea: replace expensive operations (multiplies, divides) by cheap ones (adds and subtracts)
- For loops, create a *dependent induction variable*:

- Example:

```
for (int i = 0; i<n; i++) { a[i*3] = 1; }  // stride by 3
```

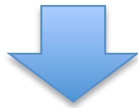


```
int j = 0;
for (int i = 0; i<n; i++) {
  a[j] = 1;
  j = j + 3;      // replace multiply by add
}
```

# Loop Unrolling (revisited)

- Branches can be expensive, unroll loops to avoid them.

```
for (int i=0; i<n; i++) { S }
```



```
for (int i=0; i<n-3; i+=4) {S;S;S;S};
for (       ; i<n; i++) { S } // left over iterations
```

- With k unrollings, eliminates (k-1)/k conditional branches
  - So for the above program, it eliminates ¾ of the branches
- Space-time tradeoff:
  - Not a good idea for large S or small n
- Interacts with instruction caching, branch prediction

# Motivating Code Analyses

- There are lots of things that might influence the safety/ applicability of an optimization
  - What algorithms and data structures can help?

- How do you know what is a loop?
- How do you know an expression is invariant?
- How do you know if an expression has no side effects?
- How do you keep track of where a variable is defined?
- How do you know where a variable is used?
- How do you know if two reference values may be aliases of one another?

# Moving Towards Register Allocation

- The OAT compiler currently generates as *many* temporary variables as it needs
  - These are the **svar**s you should be very familiar with by now.
- Current compilation strategy:
  - Each **svar** maps to a stack slot.
  - This yields programs with many loads/stores to memory.
  - Very inefficient.
- Ideally, we'd like to map as many svars as possible into registers.
  - Only 8 max registers available on 32-bit X86
  - ESP and (often) EBP are reserved, so only 6-7 really available
  - This means that a register must hold more than one svar
- When is this safe?

# Liveness

- Observation: two **`svars`** **`s1`** and **`s2`** can be assigned to the same register if their values will not be needed at the same time.
  - What does it mean for an svar to be "needed"?
  - Ans: its contents will be used as a source operand in a later instruction.
- Such a variable is called "*live*"
- Two variables can share the same register if they are not live at the same time.

# Scope vs. Liveness

- We can already get some coarse liveness information from variable scoping.
- Consider the following OAT program:

```
int f(int x) (
  int a = (int b = x + x; b * b);
  int c = a * x
)
```

- Note that due to OAT's scoping rules, variables **b** and **c** can never be live at the same time.
  - **c**'s scope is disjoint from **b**'s scope
- So, we could assign **b** and **c** to the same register
  - In fact, if you've implemented "svar recycling" in the course projects, this is one opportunity for sharing: **b** and **c** can use the same svar too.

# But Scope is too Coarse

- Consider this program:

```
int f(int x) (
  int a = x + 2;
  int b = a * a;
  int c = b + x;
  c
)
```

x is live

a and x are live

b and x are live

c is live

- The scopes of a,b,c,x all overlap – they're all in scope.
- But, a, b, c are never live at the same time.
  - So they can share the same svar / register

# Live Variable Analysis

- Note that liveness is a property of variables that holds *between* statements of the program.
- It's defined in terms of where variables are *defined* and where variables are *used*

- Liveness analysis: Compute the live variables between each statement.
  - May be *conservative* (i.e. it may claim a variable is live when it isn't) that's safe approximation
  - To be useful, it should be more *precise* than simple scoping rules.