CSEP 590 – Programming Systems
University of Washington

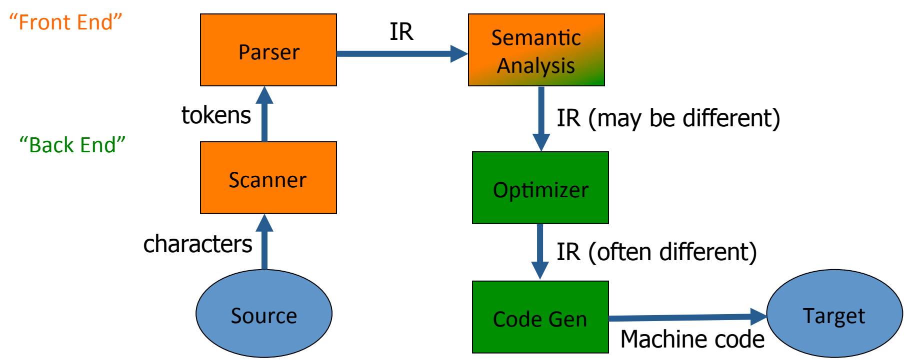
Lecture 2: Overview Part II – Back End

Michael Ringenburg
Spring 2017

Course News

- Submit presentation topic proposals by April 14
  - If you would like to work with a partner, both of you will have to present, and I will expect a more in depth/longer presentation
  - We're up to 19 students – tricky to fit >18 into final 3 weeks. Let me know if you'd be willing to present May 9.
    - Otherwise may have to come early or stay late one class (we'll vote)
- Today and next week:
  - Finish compiler overview
  - Cover 1 or 2 advanced topics in compilers:
    - Register allocation via graph coloring
    - Possibly SSA form
- After that, broaden our horizons a bit and look at other types of programming systems

Reminder: Compiler Structure



Intermediate Representations

Intermediate Representations

- The parser builds an intermediate representation of the program
  - Typically an AST
- Rest of the compiler checks and transforms the IR to improve ("optimize") it, and eventually translates it to final code
  - Typically will transform initial IR to one or more lower level IRs along the way

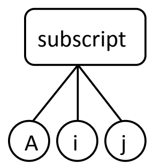
IR Design Taxonomy

- Structure
  - Graphical (trees, graphs, etc.)
  - Linear (code for some abstract machine)
  - Hybrids are common (e.g., control-flow graphs with linear code in basic blocks)
- Abstraction Level
  - High-level, near to source language
  - Low-level, closer to machine

Example: Array Reference

Source: A[i,j]

AST:



High-level linear: `t1 ← A[i,j]`

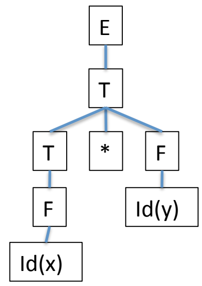
Low-level linear (3 address):

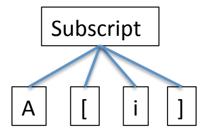
```
loadI 1      => r1
sub   rj,r1  => r2
loadI 10     => r3
mult  r2,r3  => r4
sub   ri,r1  => r5
add   r4,r5  => r6
loadI @A     => r7
add   r7,r6  => r8
load  r8     => r9
```


Graphical IRs

- IRs represented as a graph (or tree)
- Nodes and edges typically reflect some structure of the program
  - E.g., source, control flow, data dependence
- May be large (especially syntax trees)
- High-level examples:
  - Syntax trees
  - Control flow graphs
  - Data dependence graphs
  - Often used in optimization and code generation

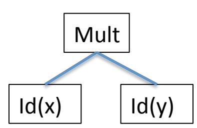
Graphical IR: Concrete Syntax Trees

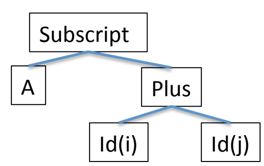




- The full grammar is needed to guide the parser, but contains many extraneous details
  - E.g., syntactic tokens, rules that control precedence
- Typically the full syntax tree does not need to be used explicitly

Graphical IR: Abstract Syntax Trees





- Want only essential structural information
- Can be represented explicitly as a tree or in a linear form, e.g., in the order of a depth-first traversal. For a[i+j], this might be:

```
Subscript
  Id(A)
  Plus
    Id(i)
    Id(j)
```

- Common output from parser; used for static semantics (type checking, etc.) and sometimes high-level optimizations

Control Flow Graph (CFG)

- Nodes are *Basic Blocks*
  - Code that always executes together (i.e., no branches into or out of the middle of the block).
  - I.e., "straightline code"
- Edges represent paths that control flow could take.
  - I.e., possible execution orderings.
  - Edge from Basic Block A to Basic Block B means Block B could execute immediately after Block A completes.
- Required for much of the analysis done in the optimizer.

CFG Example

```
print("hello");
a = 7;
if (x == y) {
  print("equal");
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print("increase");
}
print("done");
```

CFG Example

```
print("hello");
a = 7;
if (x == y) {
  print("equal");
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print("increase");
}
print("done");
```

```
print("hello");
a = 7;
if (x == y)
```


CFG Example

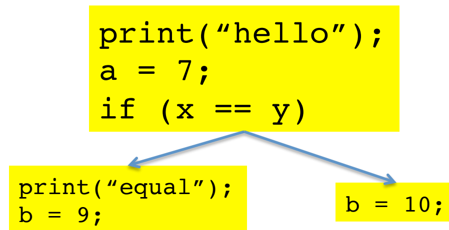
```
print("hello");
a = 7;
if (x == y) {
  print("equal");
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print("increase");
}
print("done");
```

```
print("hello");
a = 7;
if (x == y)
```

```
print("equal");
b = 9;
```

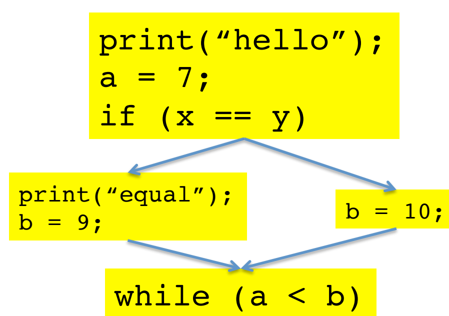
CFG Example

```
print("hello");
a = 7;
if (x == y) {
  print("equal");
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print("increase");
}
print("done");
```



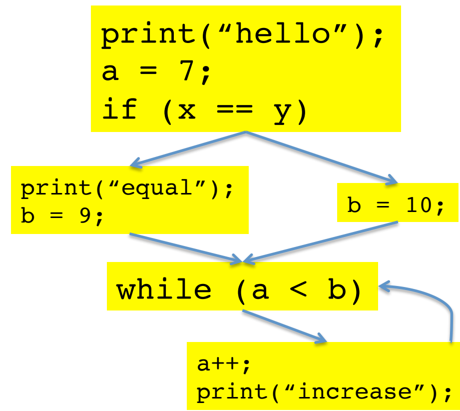
CFG Example

```
print("hello");
a = 7;
if (x == y) {
  print("equal");
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print("increase");
}
print("done");
```



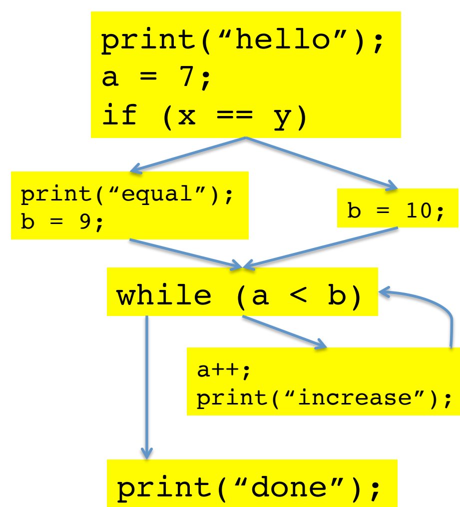
CFG Example

```
print("hello");
a = 7;
if (x == y) {
  print("equal");
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print("increase");
}
print("done");
```



CFG Example

```
print("hello");
a = 7;
if (x == y) {
  print("equal");
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print("increase");
}
print("done");
```



(Program/Data) Dependence Graph

- Often used in conjunction with another IR.
- In a data dependence graph, edges between nodes represent "dependencies" between the code represented by those nodes.
  - If A and B access the same data, and A must occur before B to achieve correct behavior, then there is a dependence edge from A to B.
  - A→B means compiler can't move B before A.
  - Granularity of nodes varies. Depends on abstraction level of rest of IR. E.g., nodes could be loads/stores, or whole statements.
  - E.g., a = 2; b = 2; c = a + 7;
    - Where's the dependence?

Types of dependencies

- Read-after-write (RAW)/"flow dependence"
  - E.g., a = 7; b = a + 1;
  - The read of 'a' must follow the write to 'a', otherwise it won't see the correct value.
- Write-after-read (WAR)/"anti dependence"
  - E.g., b = a * 2; a = 5;
  - The write to 'a' must follow the read of 'a', otherwise the read won't see the correct value.
- Write-after-write (WAW)/"output dependence"
  - E.g., a = 1; … a = 2; …
  - The writes to 'a' must happen in the correct order, otherwise 'a' will have the wrong final value.
- What about RAR/"input dependence"??

Loop-Carried Dependence

- *Loop carried dependence*: A dependence across iterations of a loop

```
for (i = 0; i < size; i++)
    x = foo(x);
```

- RAW loop carried dependence: the read of 'x' depends on the write of 'x' in the previous iteration
- Identifying and understanding these is critical for loop parallelization/vectorization
  - If the compiler "understands" the nature of the dependence, it can sometimes be removed or dealt with
  - Often use sophisticated array subscript analysis for this

Dependence Graph Example

```
a = 7;
print("hello");
while (a < b) {
  print("increase");
  a++;
}
print("done");
```

Dependence Graph Example

```
a = 7;
print("hello");
while (a < b) {
  print("increase");
  a++;
}
print("done");
```


Dependence Graph Example

```
a = 7;
print("hello");
while (a < b) {
  print("increase");
  a++;
}
print("done");
```

Dependence Graph Example

```
a = 7;
print("hello");
while (a < b) {
  print("increase");
  a++;
}
print("done");
```


Dependence Graph Example

```
a = 7;
print("hello");
while (a < b) {
  print("increase");
  a++;
}
print("done");
```

LCD

LCD: Loop-Carried Dependence

Dependence Graph Example

```
a = 7;
print("hello");
while (a < b) {
  print("increase");
  a++;
}
print("done");
```

LCD

LCD

LCD: Loop-Carried Dependence

Linear IRs

- Pseudo-code for some abstract machine
- Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked structures
- Examples: 3-address code, stack machine code

```
T1 ← 2
T2 ← b
T3 ← T1 * T2
T4 ← a
T5 ← T4 – T3
```

```
push 2
push b
multiply
push a
subtract
```

What IR to Use?

- Common choice: all(!)
  - AST or other structural representation built by parser and used in early stages of the compiler
    - Closer to source code
    - Good for semantic analysis
    - Facilitates some higher-level optimizations
  - Lower to low-level linear IR for later stages of compiler
    - Closer to machine code
    - Exposes machine-related optimizations
    - Good for resource allocation and scheduling

Semantic Analysis

What do we need to check to compile this?

```
class C {
    int a;
    C(int initial) {
     a = initial;
    }
    void setA(int val) {
     a = val;
    }
}
```

```
class Main {
  public static void
    main(String[] args) {
      C c = new C(17);
      c.setA(42);
    }
}
```


Beyond Syntax

- There is a level of correctness that is not captured by a context-free grammar
  - Has a variable been declared?
  - Are types consistent in an expression?
  - In the assignment x=y, is y assignable to x?
  - Does a method call have the right number and types of parameters?
  - In a selector p.q, is q a method or field of class instance p?
  - Is variable x guaranteed to be initialized before it is used?
  - In p.q, could p be null?
  - Etc.

Checked Properties

- Some enforced at compile time, others at run time (typically depends on language spec).
- Different languages have different requirements
  - E.g., C vs. Java typing rules, initialization requirements
  - Some of these properties are often desirable in programs, even if the language doesn't require them.
  - Compilers shouldn't enforce a property that is not required by the language (but can warn).
  - However, there are static checkers for some of these properties that use compiler-style algorithms.

What else do we need to know to generate code?

- Where are fields allocated in an object?
- How big are objects? (i.e., how much storage needs to be allocated)
- Where are local variables stored when a method is called?
  - Stack? What offset? Or exclusively in register? Which register?
- Which methods are associated with an object/class?
  - In particular, how do we figure out which method to call based on the run-time type of an object?

Semantic Analysis

- Main tasks:
  - Extract types and other information from the program
  - Check language rules that go beyond the context-free grammar
  - Resolve names
    - Relate declarations and uses of each variable
  - "Understand" the program well enough for synthesis
    - E.g., sizes, layouts of classes/structs
- Key data structure: Symbol tables
  - Map each identifier in the program to information about it (kind, type, etc.)
- This is typically considered the final part of the "front end" of the compiler (once complete, know whether or not program is legal).

Some Kinds of Semantic Information

Information	*Generated From*	*Used to process*
Symbol names (variables, methods)	Declarations	Expressions, statements
Type information	Declarations, expressions	Operations
Memory layout information	Assigned by compiler	Target code generation
Values	Constants	Expressions (constant folding)

A Sampling of Semantic Checks and Computations

- Appearance of a name in an expression: id
  - Check: Symbol has been declared and is in scope
  - Compute: Inferred type is the declared type of symbol
- Constant: v
  - Compute: Inferred type and value are explicit
  - Example: **`42.0`** has type double and value 42.0

A Sampling of Semantic Checks and Computations

- Binary operator: exp_1 op exp_2
  - Check: exp_1 and exp_2 have compatible types
    - Either identical, or well-defined conversion to appropriate types
    - Types are compatible with op
    - Example: **`42 + true`** fails, **`20 + 21.9999`** passes
  - Compute: Inferred type of expression is a function of the operator and operand types
    - Example: **`20 + 21.999`** has type double, **`42 + ", the answer"`** has type String (in Java).

Attribute Grammars

- A systematic way to think about semantic analysis
- Formalize properties checked/computed during semantic analysis and relate them to grammar productions in the CFG.
- Sometimes used directly, but even when not, AGs are a useful way to organize the analysis and think about it

Attribute Grammars

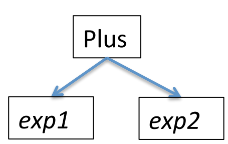
- Idea: associate attributes with each node in the syntax tree
- Examples of attributes
  - Type information
  - Storage information
  - Assignable (e.g., expression vs variable – lvalue vs rvalue for C/C++ programmers)
  - Value (for constant expressions)
  - etc. …
- Notation: X.a if a is an attribute of node X

Inherited and Synthesized Attributes

- Given a production $X ::= Y_1\ Y_2\ \ldots\ Y_n$
- A *synthesized* attribute X.a is a function of some combination of attributes of Y_i's (bottom up)
  - E.g., a value attribute
- An *inherited* attribute Y_i.b is a function of some combination of attributes X.a and other Y_j.c (top down)
  - Often restricted a bit: only Y's to the left can be used.
  - E.g., a "type environment" or a "value environment" – mappings of symbols to types or values (if they are known constants).

Attribute Equations

- For each kind of node we give a set of equations relating attribute values of the node and its children
  - Example:



plus.val = exp1.val + exp2.val

- Attribution (aka, evaluation) means implicitly finding a solution that satisfies all of the equations in the tree

Informal Example of Attribute Rules

- Suppose we have the following grammar for a trivial language
  - program ::= declList stmt
  - declList ::= declList decl | decl
  - twostmts ::= stmt stmt
  - decl ::= int id;
  - stmt ::= exp = exp ;
  - exp ::= id | exp + exp | INTEGER_LITERAL
- We want to give suitable attributes for basic type and lvalue/rvalue checking, and constant folding

Informal Example of Attribute Rules

- Attributes of nodes
  - env (type environment) stores the types of all declared variables; synthesized by declarations, inherited by the statement
    - Each entry maps a name to its type
  - envPre (for declarations) – Used to build up the environment
    - Represents the environment prior to the declaration.
    - E.g., “int x; int y;”. The envPre of “int y” will map x to an int. The env of “int y” will map x to int and y to int.
  - type (for expressions); synthesized from children (and possible env lookup)
  - kind: var (assignable) or val (not assignable); synthesized
  - value (for expressions): UNK (unknown) or an Integer, represents computed constant value; synthesized

Attributes for Declarations

- decl ::= int id;
  - decl.env = decl.preEnv U {(id, int)}
  - Intuition: add (id, int) mapping to an environment containing mappings for previous declarations
- Example: Attribution for `int y`, given that we previously saw `int x`
  - Saw `int x` earlier, so assume decl.preEnv = {(x, int)}
  - decl :: = `int y;`
  - decl.env = decl.preEnv U {(y, int)} =
    {(x, int)} U {(y, int)} =
    {(x, int), (y, int)}

Attributes for Declarations

- declList_1 ::= declList_2 decl
  - decl.preEnv = declList_2.env
  - declList_1.env = decl.env
  - Intuition: declList_2.env contains all of the previously seen mappings, so use it as the pre-environment for our new declaration. The environment for the combined list (list 1) will be the result of adding the mapping for decl to the mappings of the sublist (list 2).

Attributes for Declarations

- declList ::= decl
  - decl.preEnv = { }
  - declList.env = decl.env
  - Intuition: For the first element in our declaration list, we can start with an empty environment, because we won't have seen any declarations yet. (True here, but probably not in a real language.)

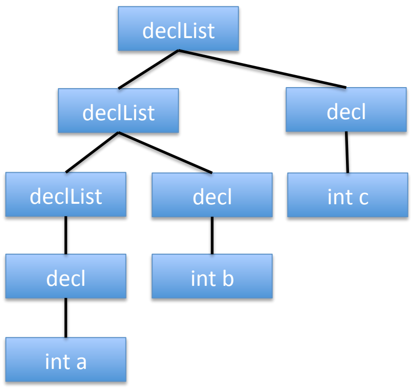
Example Declaration List

```
int a; int b; int c;
```

- declList ::= decl
  - decl.preEnv = { }
  - declList.env = decl.env
- declList_1 ::= declList_2 decl
  - decl.preEnv = declList_2.env
  - declList_1.env = decl.env
- decl ::= int id;
  - decl.env = decl.preEnv U {(id, int)}

Example Declaration List

`int a; int b; int c;`

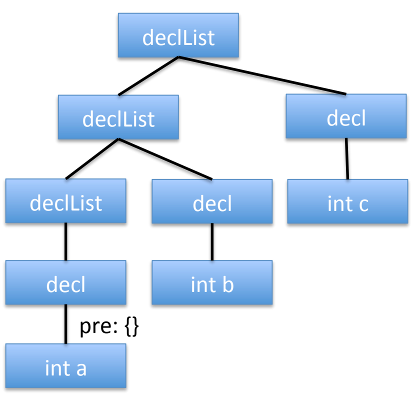


- declList ::= decl
    - decl.preEnv = { }
    - declList.env = decl.env
- declList_1 ::= declList_2 decl
    - decl.preEnv = declList_2.env
    - declList_1.env = decl.env
- decl ::= int id;
    - decl.env = decl.preEnv U {(id, int)}

Spring 2017

Example Declaration List

`int a; int b; int c;`

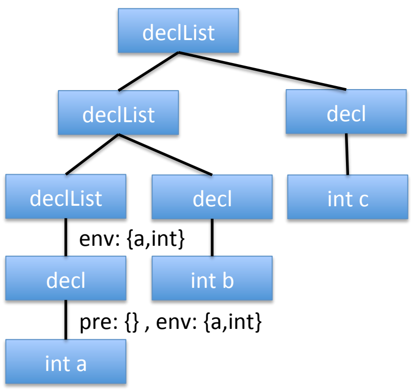


- declList ::= decl
    - decl.preEnv = { }
    - declList.env = decl.env
- declList_1 ::= declList_2 decl
    - decl.preEnv = declList_2.env
    - declList_1.env = decl.env
- decl ::= int id;
    - decl.env = decl.preEnv U {(id, int)}

Spring 2017

Example Declaration List

```
int a; int b; int c;
```

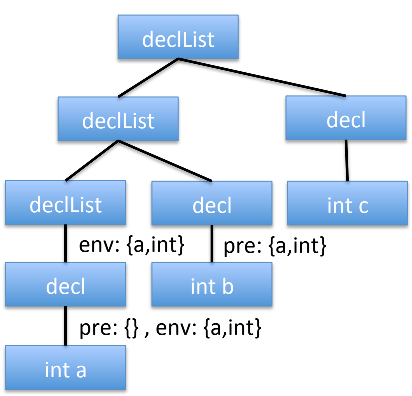


- declList ::= decl
  - decl.preEnv = { }
  - declList.env = decl.env
- declList_1 ::= declList_2 decl
  - decl.preEnv = declList_2.env
  - declList_1.env = decl.env
- decl ::= int id;
  - decl.env = decl.preEnv U {(id, int)}

Spring 2017

Example Declaration List

```
int a; int b; int c;
```

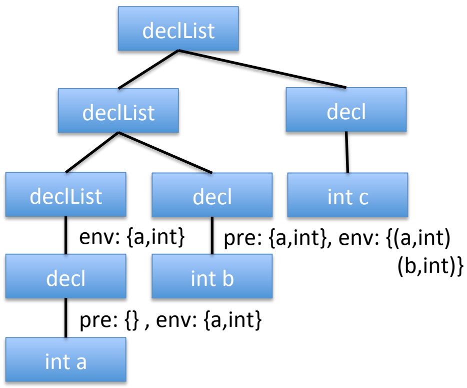


- declList ::= decl
  - decl.preEnv = { }
  - declList.env = decl.env
- declList_1 ::= declList_2 decl
  - decl.preEnv = declList_2.env
  - declList_1.env = decl.env
- decl ::= int id;
  - decl.env = decl.preEnv U {(id, int)}

Spring 2017

Example Declaration List

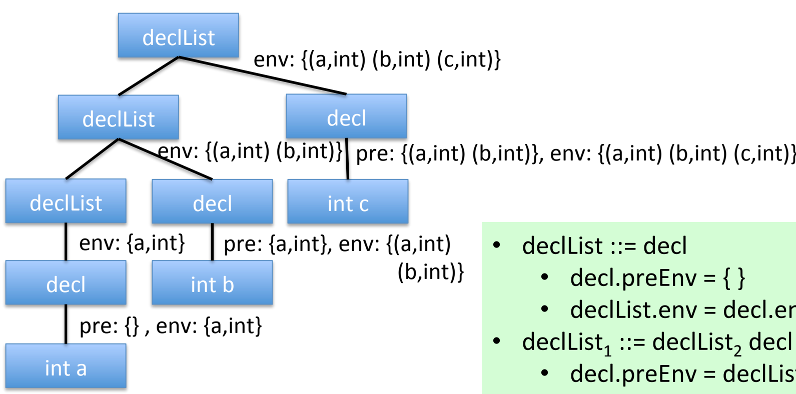
`int a; int b; int c;`



- declList ::= decl
  - decl.preEnv = { }
  - declList.env = decl.env
- declList_1 ::= declList_2 decl
  - decl.preEnv = declList_2.env
  - declList_1.env = decl.env
- decl ::= int id;
  - decl.env = decl.preEnv U {(id, int)}

Example Declaration List

`int a; int b; int c;`



- declList ::= decl
  - decl.preEnv = { }
  - declList.env = decl.env
- declList_1 ::= declList_2 decl
  - decl.preEnv = declList_2.env
  - declList_1.env = decl.env
- decl ::= int id;
  - decl.env = decl.preEnv U {(id, int)}

Attributes for Program

- program ::= declList stmt
  - stmt.env = declList.env
  - Intuition: We want to typecheck our statement given the type environment synthesized by our declaration list.

- Example: If program was

```
int a; int b; b = a + 1;
```

We would typecheck the assignment statement with the environment {(a, int), (b, int)}

Attributes for Constants

- exp ::= INTEGER_LITERAL
  - exp.kind = val
  - exp.type = int
  - exp.value = INTEGER_LITERAL
  - Intuition: An integer constant (literal) clearly has type int, and explicit value. You can't assign to it (5 = x is not legal), so it is a value (val) not a variable (var).

Attributes for Identifier Expressions

- exp ::= id
  - id.type = exp.env.lookup(id)
    - If this lookup fails, issue an undeclared variable error.
  - exp.type = id.type
  - exp.kind = var
  - exp.value = UNK
  - Intuition: We look up the identifier's type in the environment, and use that as the expression's type. If it doesn't exist in the environment, it must not have been declared, so it's an error. Since it is a variable, it is assignable and has unknown value.
  - Example: Typechecking `a` with environment {(a,int)} gives type int. Typechecking `b` with the same environment gives an error.

Attributes for Addition

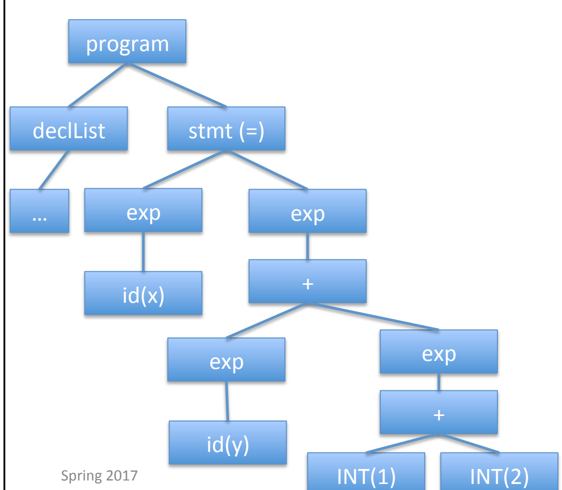
- exp ::= exp_1 + exp_2
  - exp_1.env = exp_2.env = exp.env
  - error if exp_1.type != exp_2.type (or if not compatible if using more complex rules)
  - exp.type = exp_1.type (or converted type if more complex rules)
  - exp.kind = val
  - exp.value = (exp_1.value == UNK || exp_2.value == UNK) ?
    UNK : exp_1.value + exp_2.value
  - Intuition: Typecheck operands with same environment as operation. Verify that types are compatible, and set result type appropriately. Not assignable, so set kind to val. Compute value if both operands have constant value.

Attribute Rules for Assignment

- stmt ::= exp_1 = exp_2;
  - exp_1.env = stmt.env
  - exp_2.env = stmt.env
  - Error if exp2.type is not assignment compatible with exp1.type
  - Error if exp_1.kind is not var (can't be val)
  - Intuition: Verify that left hand side is assignable, and that types of left and right hand sides are compatible.

Example

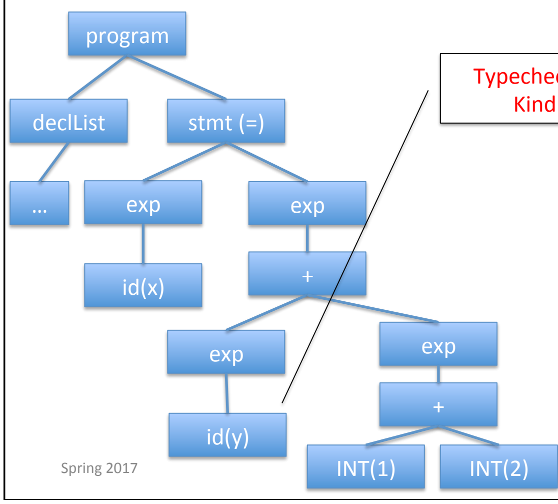
int x; int y; int z; x = y + (1+2);



- program ::= declList stmt
  - stmt.env = declList.env
- exp ::= INTEGER_LITERAL
  - exp.kind = val
  - exp.type = int
  - exp.value = INTEGER_LITERAL
- exp ::= id
  - type = exp.env.lookup(id)
    - (error if fails)
  - exp.type = id.type
  - exp.kind = var
  - exp.value = UNK

Example

int x; int y; int z; x = y + (1+2);



Typecheck y with declList env
Kind: var, Value: UNK

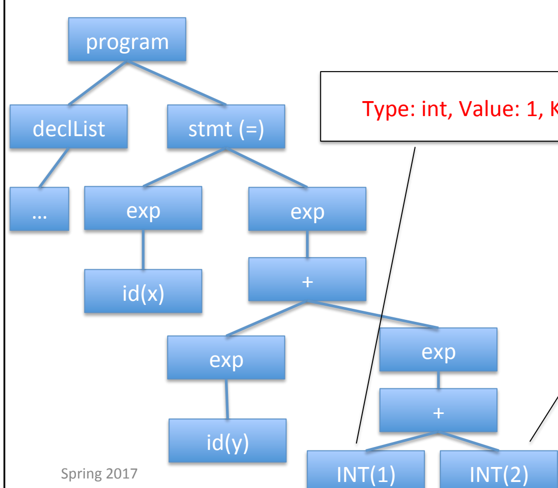
- program ::= declList stmt
  - stmt.env = declList.env
- exp ::= INTEGER_LITERAL
  - exp.kind = val
  - exp.type = int
  - exp.value = INTEGER_LITERAL
- exp ::= id
  - type = exp.env.lookup(id)
    - (error if fails)
  - exp.type = id.type
  - exp.kind = var
  - exp.value = UNK

Example

int x; int y; int z; x = y + (1+2);

Type: int, Value: 2, Kind: val

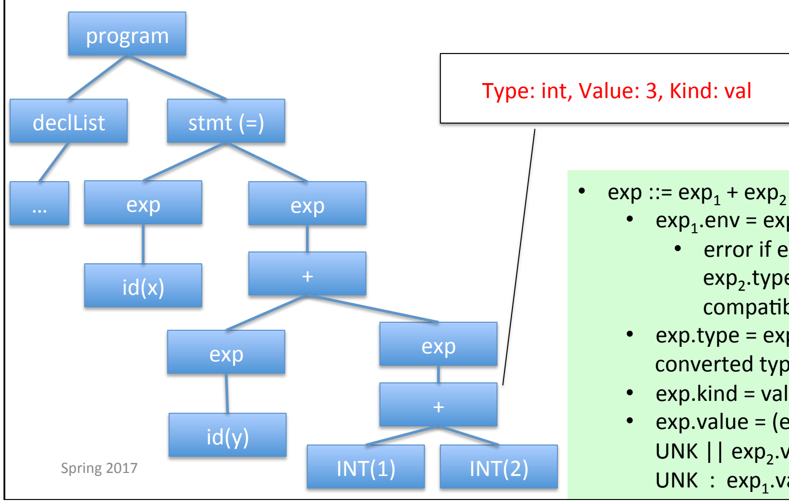
Type: int, Value: 1, Kind: val



- program ::= declList stmt
  - stmt.env = declList.env
- exp ::= INTEGER_LITERAL
  - exp.kind = val
  - exp.type = int
  - exp.value = INTEGER_LITERAL
- exp ::= id
  - type = exp.env.lookup(id)
    - (error if fails)
  - exp.type = id.type
  - exp.kind = var
  - exp.value = UNK

Example

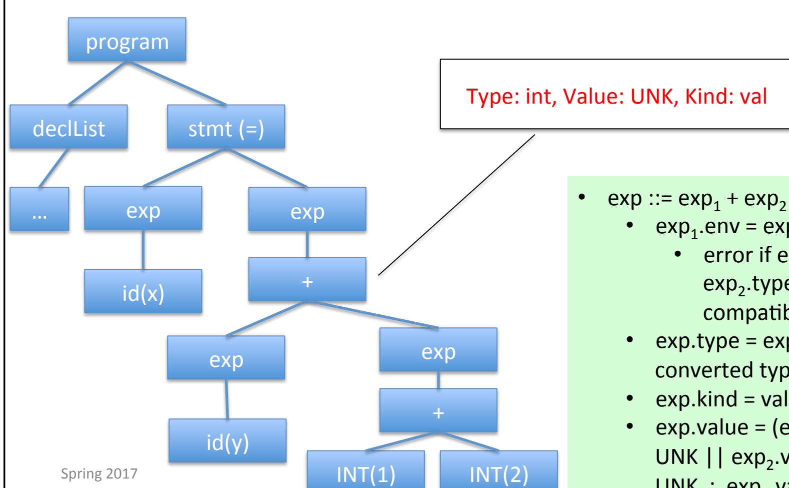
int x; int y; int z; x = y + (1+2);



- exp ::= exp_1 + exp_2
  - exp_1.env = exp_2.env = exp.env
    - error if exp_1.type != exp_2.type (or if not compatible)
  - exp.type = exp_1.type (or converted type)
  - exp.kind = val
  - exp.value = (exp_1.value == UNK || exp_2.value == UNK) ? UNK : exp_1.value + exp_2.value

Example

int x; int y; int z; x = y + (1+2);

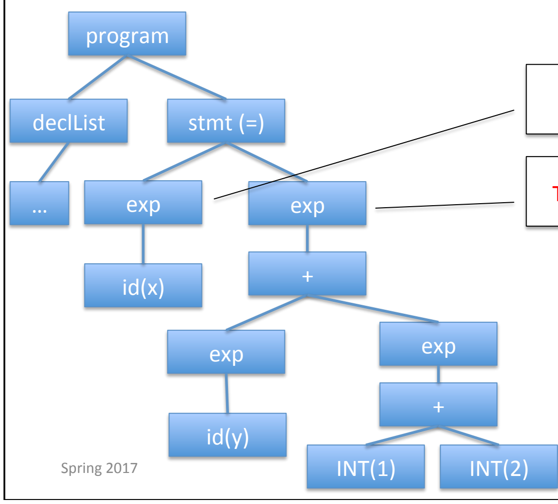


- exp ::= exp_1 + exp_2
  - exp_1.env = exp_2.env = exp.env
    - error if exp_1.type != exp_2.type (or if not compatible)
  - exp.type = exp_1.type (or converted type)
  - exp.kind = val
  - exp.value = (exp_1.value == UNK || exp_2.value == UNK) ? UNK : exp_1.value + exp_2.value

Spring 2017

Example

int x; int y; int z; x = y + (1+2);



Typecheck x with declList env
Type:int, Kind: var

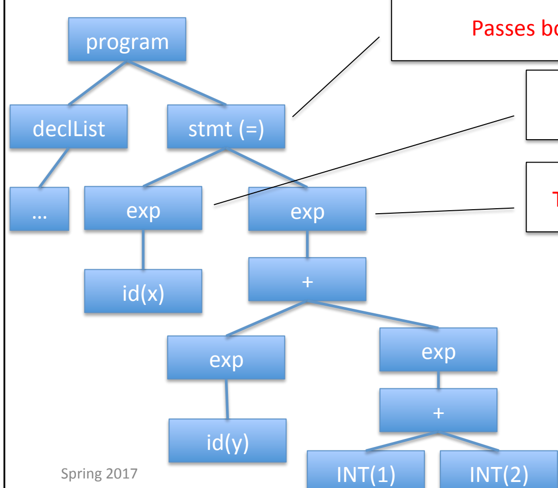
Type: int, Value: UNK, Kind: val

- stmt ::= exp_1 = exp_2;
  - exp_1.env = stmt.env
  - exp_2.env = stmt.env
  - Error if exp2.type is not assignment compatible with exp1.type
  - Error if exp_1.kind is not var (can't be val)

Example

int x; int y; int z; x = y + (1+2);

Passes both checks!



Typecheck x with declList env
Type:int, Kind: var

Type: int, Value: UNK, Kind: val

- stmt ::= exp_1 = exp_2;
  - exp_1.env = stmt.env
  - exp_2.env = stmt.env
  - Error if exp2.type is not assignment compatible with exp1.type
  - Error if exp_1.kind is not var (can't be val)

Observations

- These are equational (functional) computations
- This can be automated (if equations are non-circular)
- But implementation problems
  - Non-local computation: Attribute equations can only refer to values associated with symbols that appear in a single production rule.
    - If you need non-local values, you need to add special rules to the grammar to copy them around. Can make grammar very large.
  - Can't afford to literally pass around copies of large, aggregate structures like environments.
  - Use of production rules binds attributes to the parse tree rather than the (typically smaller, and more useful) AST. Can work around this (use "AST grammar"), but results in more complex attribute rules.

In Practice

- Attribute grammars give us a good way of thinking about how to structure semantic checks
- Symbol tables will hold environment information
- Add fields to AST nodes to refer to appropriate attributes (symbol table entries for identifiers, types for expressions, etc.)
  - Put in appropriate places in AST class inheritance tree – most statements don't need types, for example

Symbol Tables

- Map identifiers to
  <type, kind, location, other properties>
- Operations
  - Lookup(id) => information
  - Enter(id, information)
  - Open/close scopes
- Build & use during semantics pass
  - Build first from declarations
  - Then use to check semantic rules
- Use (and add to) during later phases as well

Code Generation

Basic Code Generation Strategy

- Walk the AST or other IR, outputting code for each construct encountered
- Handling of node's children is dependent on type of node
  - E.g., for binary operation like +:
    - Generate code to compute operand 1 (and store result)
    - Generate code to compute operand 2 (and store result)
    - Generate code to load operand results and add them together
- Today is just a sampling of basic constructs, to give basic idea

Conventions for Examples

- The following slides will walk through how this is done for many common language constructs
- Examples show code snippets in isolation
- Register eax used below as a generic example
  - Rename as needed for more complex code using multiple registers
- A few *peephole optimizations* included below for a flavor of what's possible
  - Localized optimizations performed on small ASM instruction sequences.

Variables

- For our purposes, assume all data will be in either:
  - A stack frame (method local variables)
  - An object (instance variables)
- Local variables accessed via ebp (stack base pointer)

```
mov  eax,[ebp-12]
```

- Object instance variables accessed via an object address in a register

Code Generation for Constants

- Source

```
17
```

- x86

```
mov   eax,17
```

  - Idea: realize constant value in a register

- Optimization: if constant is 0

```
xor   eax,eax
```

  - May be smaller and faster

Assignment Statement

- Source

  var = exp;

- x86

  <code to evaluate exp into, say, eax>

  mov [ebp+offset$_{var}$],eax

Unary Minus

- Source

  -exp

- x86

  <code evaluating exp into eax>

  neg eax

- Optimization
  - Collapse -(-exp) to exp
- Unary plus is a no-op

Binary +

- Source

    exp1 + exp2

- x86

    <code evaluating exp1 into eax>

    <code evaluating exp2 into edx>

    add eax,edx

Binary +

- Optimizations
  - If exp2 is a simple variable or constant, don't need to load it into another register first. Instead:

    add eax,imm_{Const} ; imm is constant

    add eax,[ebp+$offset_{var}$] ; offset is variable's stack offset

  - Change exp1 + (-exp2) into exp1-exp2
  - If exp2 is 1

    inc eax

    - Somewhat surprising: whether this is better than add eax,1 depends on processor implementation and has changed over time

Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos
- In the following, j_{false} is used to mean jump when a condition is false
  - No such instruction on x86
  - Can realize with appropriate sequence of instructions to set condition codes followed by conditional jumps
  - Normally don't actually generate the value "true" or "false" in a register

While

- Source

  while (cond) stmt

- X86

  test: <code evaluating cond>
  j_{false} done
  <code for stmt>
  jmp test
  done:

  - Note: In generated asm code we'll need to generate *unique* labels for each loop

If

- Source

    if (cond) stmt

- x86

    <code evaluating cond>

    j_{false} skip

    <code for stmt>

  skip:

Boolean Expressions

- What do we do with this?

    x > y

- It is an expression that evaluates to true or false
  - Could generate the value (0/1 or whatever the local convention is)
  - But normally we don't want/need the value; we're only trying to decide whether to jump
    - One exception: assignment expressions, e.g.,

      while (my_bool = (x < y)) { … }

Code for exp1 > exp2

- Generated code depends on context
  - What is the jump target?
  - Jump if the condition is true or if false?
- Example: evaluate exp1 > exp2, jump on false, target if jump taken is L123

```
<evaluate exp1 to eax>
<evaluate exp2 to edx>
cmp  eax,edx
jng   L123     ; greater-than test, jump on false, so jng
               ; (jump not greater)
```


Optimization Overview

Optimizations

- Use added passes to identify inefficiencies in intermediate or target code
- Replace with equivalent ("has the same externally visible behavior") but better sequences
  - Better can mean many things: faster, smaller, less memory, more energy-efficient, etc.
- Target-independent optimizations best done on IR code
  - Removing redundant computations, dead code, etc.
- Target-dependent optimizations best done on target code
  - Generating sequence that are more efficient on a particular machine
- "Optimize" overly optimistic: "usually improve" is generally more accurate
  - And "clever" programmers can outwit you!

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 = *(fp + ioffset);   // i
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);   // a[i]
t5 = 2;
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);   // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset); // i
t14 = t13 * 4;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Strength Reduction: shift often cheaper than multiply

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = t5 << 2;
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Constant propagation: Replace variables with known constant value.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = 2 << 2; // was t5 << 2
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - 5;  // was t10 - t11
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Dead Store (or Dead Assignment) Elimination: Remove assignements to provably unused variables.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = 2 << 2;
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t11 = 5;
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 88

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Dead Store (or Dead Assignment) Elimination: Remove stores to provably unused variables.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t6 = 2 << 2;
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 89

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Constant Folding: Statically compute operations with only constant operands.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t6 = 8; // was 2 << 2
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 90

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Constant Propagation, then Dead Store Elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t6 = 8;
t7 = fp + 8; // was fp + t6
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 91

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Constant Propagation, then Dead Store Elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = fp + 8;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Applying arithmetic identities: We know + is commutative & associative. boffset is typically a known compile-time constant (say, -30), so this enables …

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = boffset + 8;
t8 = *(t7 + fp);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

… more constant folding.
Which in turn enables …

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = -22; // was boffset(-30) + 8
t8 = *(t7 + fp);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

More constant propagation
and dead store elimination.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t7 = -22;
t8 = *(fp - 22);  // b[2] (was t7+fp)
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

More constant propagation and dead store elimination.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = *(fp + ioffset); // i
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Common subexpression elimination: No need to compute *(fp+ioffset) twice if we know it won't change.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = *(fp + xoffset); // x
t12 = t10 - 5;
t13 = t1; // i   (was *(fp+ioffset))
t14 = t13 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Copy propagation: Replace assignment targets with their values. E.g., replace t13 with t1.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = t9; // x  (was *(fp+xoffset))
t12 = t10 - 5;
t13 = t1; // i
t14 = t1 << 2; // was t13 << 2
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

More copy propagation

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = t9; // x
t12 = t9 - 5;  // Was t10 - 5
t13 = t1; // i
t14 = t1 << 2;
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Common subexpression elimination.

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = t9; // x
t12 = t9 – 5;
t13 = t1; // i
t14 = t2;  // was t1 << 2
t15 = fp + t14;
*(t15 + coffset) = t12; // c[i] := …
```


An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Copy Propagation

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t10 = t9; // x
t12 = t9 – 5;
t13 = t1; // i
t14 = t2;
t15 = fp + t2; // was fp + t14
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Dead Assignment Elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
```
~~t10 = t9; // x~~
```
t12 = t9 - 5;
```
~~t13 = t1; // i~~
~~t14 = t2;~~
```
t15 = fp + t2;
*(t15 + coffset) = t12; // c[i] := …
```


An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

Dead Assignment Elimination

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t12 = t9 - 5;
t15 = fp + t2;
*(t15 + coffset) = t12; // c[i] := …
```

An example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 22);  // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9; // x = …
t12 = t9 – 5;
t15 = fp + t2;
*(t15 + coffset) = t12; // c[i] := …
```

- Final: 3 loads (i, a[i], b[2]), 2 stores (x, c[i]), 5 register-only moves, 9 +/-, 1 shift
- Original: 5 loads, 2 stores, 10 register-only moves, 12 +/-, 3 *
  - (Optimizer typically deals in "pseudo-registers" – can have as many as you want – and lets register allocator figure out optimal assignments of pseudo-registers to real registers.)

Kinds of Optimizations

- peephole: look at adjacent instructions
- local: look at individual *basic blocks*
  – Straight-line sequence of statements
- intraprocedural: look at whole procedure
  – Commonly called "global"
- interprocedural: look across procedures
  – "whole program" analysis
  – gcc's "link time optimization" is a version of this
- Larger scope => usually better optimization but more cost and complexity
  – Analysis is often less precise because of more possibilities

Peephole Optimization

- After target code generation, look at adjacent instructions (a "peephole" on the code stream)
  - try to replace adjacent instructions with something faster, e.g., store and load with store and register move:

`movq %r9,12(%rsp)` `movq 12(%rsp),%r12`	`movq %r9,12(%rsp)` `movq %r9,%r12`

  - Jump chaining can also be considered a form of peephole optimization (removing jump-to-jump)

Algebraic Simplification

- "constant folding": pre-calculate operation on constant
- "strength reduction": replace operation with a cheaper operation
- "simplification": applying algebraic identities
  - `z = 3 + 4;` → `z = 7;`
  - `z = x + 0;` → `z = x;`
  - `z = x * 1;` → `z = x;`
  - `z = x * 2;` → `z = x << 1;` or `z = x + x;`
  - `z = x * 8;` → `z = x << 3;`
  - `z = x / 8;` → `z = x >> 3;`
  - `z = (x + y) - y;` → `z = x;`
- Can be done at many levels, from peephole on up.
- Why do these examples happen?
  - Often created: Conversion to lower-level IR, Other optimizations, Code generation

Higher-level Example: Loop-based Strength Reduction

```
for (int i = 0; i < size; i++) {
  foo[i] = i;
}
```

```
for (int i = 0; i < size; i++) {
  *(foo + i * elementSize) = i;
}
```

```
t1 = 0;
for (int i = 0; i < size; i++) {
  *(foo + t1) = i;
  t1 = t1 + 8;
}
```

- Sometimes multiplication by the loop variable in a loop can be replaced by additions into a temporary accumulator
- Similarly, exponentiation can be replaced by multiplication.

Local Optimizations

- Analysis and optimizations within a basic block
- *Basic block*: straight-line sequence of statements
  - no control flow into or out of middle of sequence
- Not too hard to implement with a reasonable IR

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable is next assigned)
- Can enable more constant folding
  - Code; unoptimized intermediate code:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = count;`
	`t2 = 5;`
	`t3 = t1 * t2;`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4, t5);`
	`y = t6;`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable is next assigned)
- Can enable more constant folding
  - Code; propagated intermediate code:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;    // CP count`
	`t2 = 5;`
	`t3 = 10 * 5;  // CP t1`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4, 3); // CP t5`
	`y = t6;`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable is next assigned)
- Can enable more constant folding
  - Code; folded intermediate code:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;    // CF 5 * 10`
	`x = t3;`
	`t4 = x;`
	`t5 = 3;`
	`t6 = exp(t4, 3);`
	`y = t6;`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable is next assigned)
- Can enable more constant folding
  - Code; repropagated intermediate code:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;   // CP t3`
	`t4 = 50;  // CP x`
	`t5 = 3;`
	`t6 = exp(50, 3);   // CP t4`
	`y = t6;`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable is next assigned)
- Can enable more constant folding
  - Code; refolded intermediate code:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;`
	`t4 = 50;`
	`t5 = 3;`
	`t6 = 125000;   // CF 50^3`
	`y = t6;`

Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant (until variable is next assigned)
- Can enable more constant folding
  - Code; repropagated intermediate code:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;`
	`t4 = 50;`
	`t5 = 3;`
	`t6 = 125000;`
	`y = 125000;   // CP t6`

Local Dead Assignment Elimination

- If left side of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen?
  - Clean-up after previous optimizations, often
- Intermediate code after constant propagation:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	`t1 = 10;`
	`t2 = 5;`
	`t3 = 50;`
	`x = 50;`
	`t4 = 50;`
	`t5 = 3;`
	`t6 = 125000;`
	`y = 125000;  // CP t6`

Local Dead Assignment Elimination

- If left side of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen?
  - Clean-up after previous optimizations, often
- Intermediate code after constant propagation:

`count = 10;`
`... // No count assigns`
`x = count * 5;`
`y = x ^ 3;`
	`count = 10`
	~~`t1 = 10;`~~
	~~`t2 = 5;`~~
	~~`t3 = 50;`~~
	`x = 50;`
	~~`t4 = 50;`~~
	~~`t5 = 3;`~~
	~~`t6 = 125000;`~~
	`y = 125000;  // CP t6`

Local Common Subexpression Elimination

- Looks for repetitions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk basic block, keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = *(fp + ioffset);`
	`t6 = t5 * 4;`
	`t7 = fp + t6;`
	`t8 = *(t7 + boffset);`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Looks for repitiions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk basic block, keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = t1;  // CSE`
	`t6 = t5 * 4;`
	`t7 = fp + t6;`
	`t8 = *(t7 + boffset);`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk basic block, keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = t1;`
	`t6 = t1 * 4;   // CP`
	`t7 = fp + t6;`
	`t8 = *(t7 + boffset);`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk basic block, keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = t1;`
	`t6 = t2;  // CSE`
	`t7 = fp + t2; // CP`
	`t8 = *(t7 + boffset);`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk basic block, keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	`t5 = t1;`
	`t6 = t2;`
	`t7 = t3; // CSE`
	`t8 = *(t3 + boffset);//CP`
	`t9 = t4 + t8;`

Local Common Subexpression Elimination

- Looks for repititions of the same computation, and eliminate them if the result won't have changed (and no side effects)
  - Avoids repeating the same calculation
  - Eliminates redundant loads
- Idea: walk basic block, keeping track of available expressions

`... a[i] + b[i] ...`	`t1 = *(fp + ioffset);`
	`t2 = t1 * 4;`
	`t3 = fp + t2;`
	`t4 = *(t3 + aoffset);`
	~~t5 = t1;~~ `// DAE`
	~~t6 = t2;~~ `// DAE`
	~~t7 = t3;~~ `// DAE`
	`t8 = *(t3 + boffset);`
	`t9 = t4 + t8;`

Intraprocedural optimizations

- Enlarge scope of analysis to whole procedure
  - more opportunities for optimization
  - have to deal with branches, merges, and loops
- Can do constant propagation, common subexpression elimination, etc. at function-wide level
- Can do new things, e.g. loop optimizations
- Optimizing compilers usually work at this level (-O2)

Code Motion

- Goal: move loop-invariant calculations out of loops
- Can do at source level or at intermediate code level

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + b[j];
  z = z + (foo*bar)^2;
}
```

```
t1 = b[j];
t2 = (foo*bar)^2;
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + t1;
  z = z + t2;
}
```

Interprocedural Optimization

- Expand scope of analysis to procedures calling each other
- Can do local & intraprocedural optimizations at larger scope
- Can do new optimizations, e.g. inlining

Inlining: replace call with body

- Replace procedure call with body of called procedure, and substituting actual arguments for formal parameters
- Source:

```
final double pi = 3.1415927;
double circle_area(double radius) {
   return pi * (radius * radius);
}
...
double r = 5.0;
...
double a = circle_area(r);
```

- After inlining:

```
double r = 5.0;
...
double a = pi * r * r;
```

- (Then what? Constant propagation/folding.)

Data Structures for Optimizations

- Need to represent control and data flow
- Control flow graph (CFG) captures flow of control
  - nodes are basic blocks
  - edges represent (all possible) control flow
  - node with multiple successors = branch/switch
  - node with multiple predecessors = merge or join point
  - loop in graph = loop
- Data flow graph (DFG) capture flow of data, e.g. def/use chains:
  - nodes are def(inition)s and uses of data/variables
  - edges from defs to uses of (potentially) the same data
  - a def can reach multiple uses
  - a use can have multiple reaching defs (different control flow, possible aliasing, etc.)

Analysis and Transformation

- Each optimization is made up of
  - some number of analyses
  - followed by a transformation
- Analyze CFG and/or DFG by propagating info forward or backward along CFG and/or DFG edges
  - merges in graph require combining info
  - loops in graph require (conservative) *iterative approximation*
- Perform (improving) transformations based on info computed
- Analysis must be conservative/safe/sound so that transformations preserve program behavior

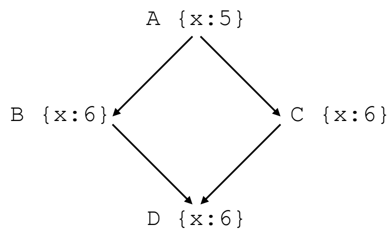
Example: Constant Propagation, Folding

- Can use either the CFG or the DFG
- CFG analysis info: table mapping each variable in scope to one of:
  - a particular constant
  - NonConstant
  - Undefined
- Transformation at each instruction:
  - If encounter an assignment of a constant to a variable, set variable as constant
  - if reference a variable that the table maps to a constant, then replace with that constant (constant propagation)
  - if r.h.s. expression involves only constants, and has no side-effects, then perform operation at compile-time and replace r.h.s. with constant result (constant folding)
- For best analysis, do constant folding as part of analysis, to learn all constants in one pass

Merging data flow analysis info

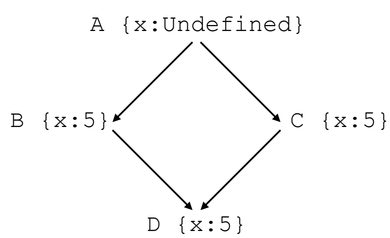
- Constraint: merge results must be sound
  - if something is believed true after the merge, then it must be true no matter which path we took into the merge
  - only things true along all predecessors are true after the merge
- To merge two maps of constant information, build map by merging corresponding variable information
- To merge information about two variable
  - if one is Undefined, keep the other (uninitialized variables in many languages allowed to have any value)
  - if both are the **same** constant, keep that constant
  - otherwise, degenerate to NonConstant

Example Merges



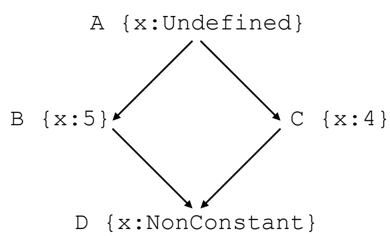

```
// Block A
int x;
x = 5;
if (foo) {
  // Block B
  x++;
} else {
  // Block C
  x=6;
}
// Block D
…
```


Example Merges



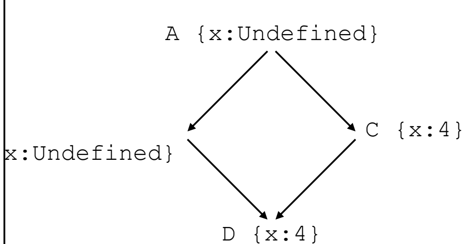

```
// Block A
int x;
if (foo) {
  // Block B
  z++;
  x = 5;
} else {
  // Block C
  z--;
  x = 5;
}
// Block D
…
```

Example Merges




```
// Block A
int x;
if (foo) {
  // Block B
  z++;
  x = 5;
} else {
  // Block C
  z--;
  x = 4;
}
// Block D
…
```


Example Merges



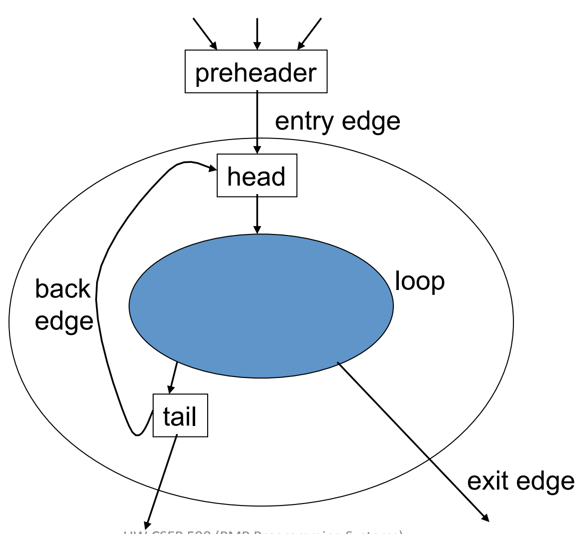

```
// Block A
int x;
if (foo) {
  // Block B
  z++;
} else {
  // Block C
  z--;
  x = 4;
}
// Block D
…
```

How to analyze loops

```
i = 0;
x = 10;
y = 20;
while (...) {
  // what's true here?
  ...
  i = i + 1;
  y = 30;
}
// what's true here?
... x ... i ... y ...
```

- Safe but imprecise: forget everything when we enter or exit a loop
- Precise but unsafe: keep everything when we enter or exit a loop
- Can we do better?

Loop Terminology



Optimistic Iterative Analysis

- Assuming information at loop head is same as information at loop entry
- Then analyze loop body, computing information at back edge
- Merge information at loop back edge and loop entry
- Test if merged information is same as original assumption
  - If so, then we're done
  - If not, then replace previous assumption with merged information,
  - and go back to analysis of loop body

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
    // what's true here?
    ...
    i = i + 1;
    y = 30; }
// what's true here?
... x ... i ... y ...
```

i = 0, x = 10, y = 20

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
    // what's true here?
    ...
    i = i + 1;
    y = 30; }
// what's true here?
... x ... i ... y ...
```

i = 0, x = 10, y = 20

i = 1, x = 10, y = 30

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

i = NC, x = 10, y = NC

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

i = NC, x = 10, y = NC

i = NC, x = 10, y = NC

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

i = NC, x = 10, y = NC

i = NC, x = 10, y = NC

Why does this work?

- Why are the results always conservative?
- Because if the algorithm stops, then
  - the loop head info is at least as conservative as both the loop entry info and the loop back edge info
  - the analysis within the loop body is conservative, given the assumption that the loop head info is conservative

Optimization Summary

- Optimizations organized as collections of passes, each rewriting IL in place into (hopefully) better version
- Each pass does analysis to determine what is possible, followed by (or concurrent with) transformations that (hopefully) improve the program
  - Sometimes have "analysis-only" passes – produce info used by later passes

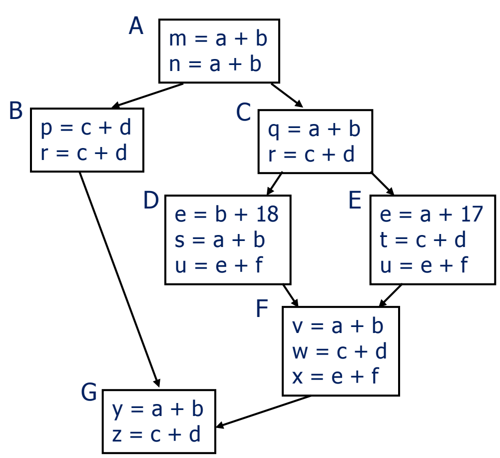
Dataflow Analysis
(if we have extra time and energy!)

Next topic: Dataflow Analysis

- A framework and algorithm for many common compiler analyses
- Initial example: dataflow analysis for common subexpression elimination
- Other analysis problems that work in the same framework
- We'll be discussing some of the same optimizations we saw in the optimization overview, but with more formalism and details.

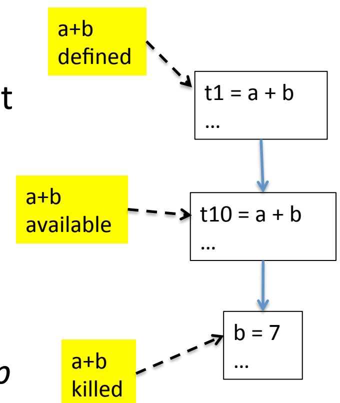
Motivating Example: Common Subexpression Elimination (CSE)

- Goal: Find common subexpressions, replace with temporaries
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression – copy a temp instead
  – Simple inside a single block; more complex dataflow analysis used across bocks



"Available" and Other Terms

- An expression *e* is *defined* at point *p* in the CFG (control flow graph) if its value is computed at *p*
  - Sometimes called *definition site*
- An expression *e* is *killed* at point *p* if one of its operands (components) is redefined at *p*
  - Sometimes called *kill site*
- An expression *e* is *available* at point *p* if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*



Available Expression Sets

- To compute available expressions, for each block *b*, define
  - AVAIL(b) – the set of expressions available on entry to *b*
  - NKILL(b) – the set of expressions <u>not killed</u> in *b*
  - DEF(b) – the set of expressions defined in *b* and not subsequently killed in *b*

Computing Available Expressions

- AVAIL(b) (expressions available on entry to b) is the set

  $\text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$

  – preds(b) is the set of b's direct predecessors in the CFG
  – In "english", the expressions available on entry to b are the expressions that were available at the end of *every* directly preceding basic block x. (This is the $\bigcap_{x \in \text{preds}(b)}$)
  – The expressions available at the end of block x are exactly those that were defined in x (and not killed), and those that were available at the beginning of x and not killed in x.
- Applying to every block gives a system of simultaneous equations – a dataflow problem

Computing Available Expressions

- Big Picture
  – Build control-flow graph
  – Calculate initial local data – DEF(b) and NKILL(b) for every block b
    - This only needs to be done once
  – Iteratively calculate AVAIL(b) by repeatedly evaluating equations until nothing changes
    - A fixed-point algorithm

Computing DEF and NKILL (1)

- For each block *b* with operations $o_1, o_2, \ldots, o_k$

KILLED = ∅ // Killed *variables* (not expressions)
DEF(b) = ∅
for i = k to 1   // Note we are working *backwards* - important
   assume o_i is "x = y + z"
   if (y ∉ KILLED and z ∉ KILLED)   // Expression in DEF only if
      add "y + z" to DEF(b)   // they aren't later killed
   add x to KILLED
…

Example: Computing DEF and KILL

x = a + b;
b = c + d;
m = 5*n;

DEF = { }
KILL = { }

Example: Computing DEF and KILL

```
x = a + b;
b = c + d;
m = 5*n;
```

DEF = { 5*n }
KILL = { m }

Example: Computing DEF and KILL

```
x = a + b;
b = c + d;
m = 5*n;
```

DEF = { 5*n, c+d }
KILL = { m, b }

Example: Computing DEF and KILL

```
x = a + b;
b = c + d;
m = 5*n;
```

DEF = { 5*n, c+d }
KILL = { m, **b**, x }

(b is killed, so don't add a+b to DEF)

Computing DEF and NKILL (2)

- After computing DEF and KILL for a block b, conceptually we do the following:

// NKILL is expressions *not* killed.
NKILL(b) = { all expressions in program}
for each expression *e* // Remove any killed
  for each variable *v* ∈ e
    if *v* ∈ KILL then
      NKILL(b) = NKILL(b) - *e*

Example: Computing DEF and NKILL

```
x = a + b;
b = c + d;
m = 5*n;
```

DEF = { 5*n, c+d }
KILL = { m, b, x }
NKILL = all expressions that don't use m, b, or x

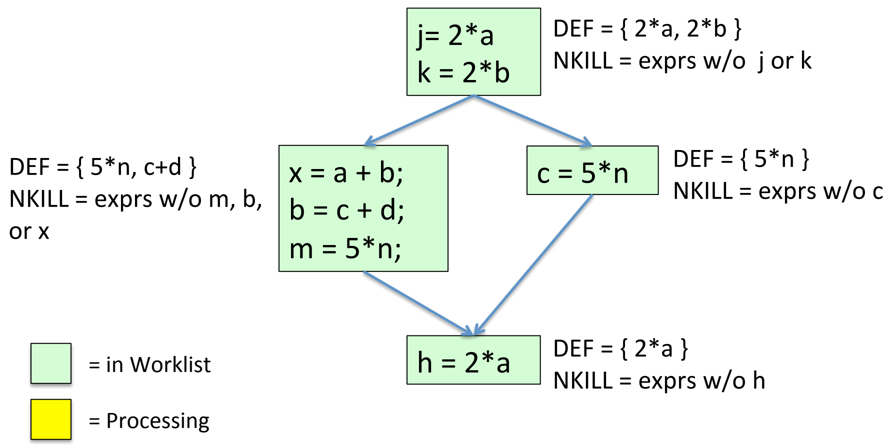
Computing Available Expressions

- Once DEF(b) and NKILL(b) are computed for all blocks b, compute AVAIL for all blocks by repeatedly applying the previous formula in a fixed-point algorithm:

Worklist = { all blocks b_i }
while (Worklist ≠ ∅)
    remove a block b from Worklist
    // If b in Worklist, at least 1 predecessor changed
    let AVAIL(b) = $\bigcap_{x \in preds(b)}$ (DEF(x) ∪ (AVAIL(x) ∩ NKILL(x)))
    if AVAIL(b) changed
        Worklist = Worklist ∪ successors(b)

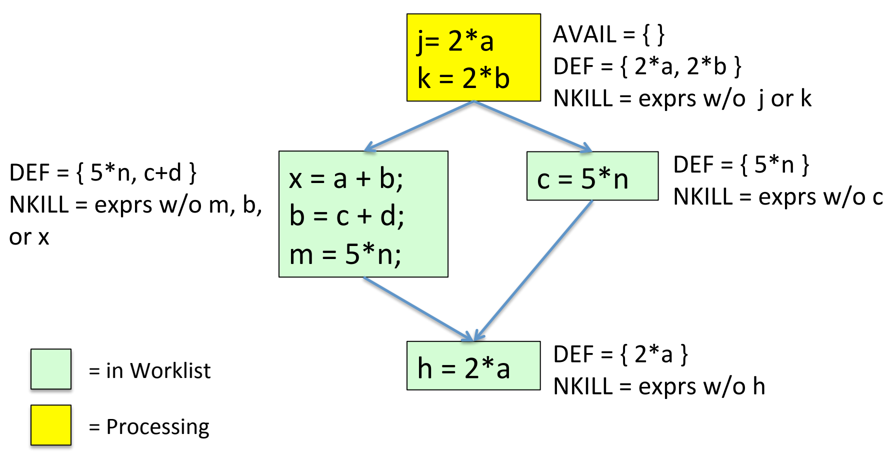
Example: Computing DEF and NKILL

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) ∪ (AVAIL(x) ∩ NKILL(x)))



Example: Computing DEF and NKILL

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) ∪ (AVAIL(x) ∩ NKILL(x)))



Example: Computing DEF and NKILL

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) ∪ (AVAIL(x) ∩ NKILL(x)))

j= 2*a
k = 2*b

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

DEF = { 5*n, c+d }
NKILL = exprs w/o m, b, or x

x = a + b;
b = c + d;
m = 5*n;

c = 5*n

DEF = { 5*n }
NKILL = exprs w/o c

h = 2*a

AVAIL = { 5*n }
DEF = { 2*a }
NKILL = exprs w/o h

= in Worklist

= Processing

Example: Computing DEF and NKILL

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) ∪ (AVAIL(x) ∩ NKILL(x)))

j= 2*a
k = 2*b

AVAIL = { }
DEF = { 2*a, 2*b }
NKILL = exprs w/o j or k

AVAIL = {2*a, 2*b}
DEF = { 5*n, c+d }
NKILL = exprs w/o m, b, or x

x = a + b;
b = c + d;
m = 5*n;

c = 5*n

DEF = { 5*n }
NKILL = exprs w/o c

h = 2*a

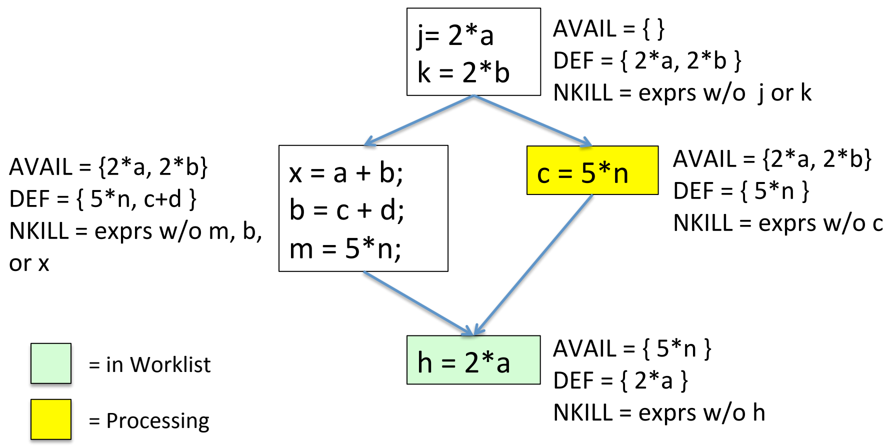
AVAIL = { 5*n }
DEF = { 2*a }
NKILL = exprs w/o h

= in Worklist

= Processing

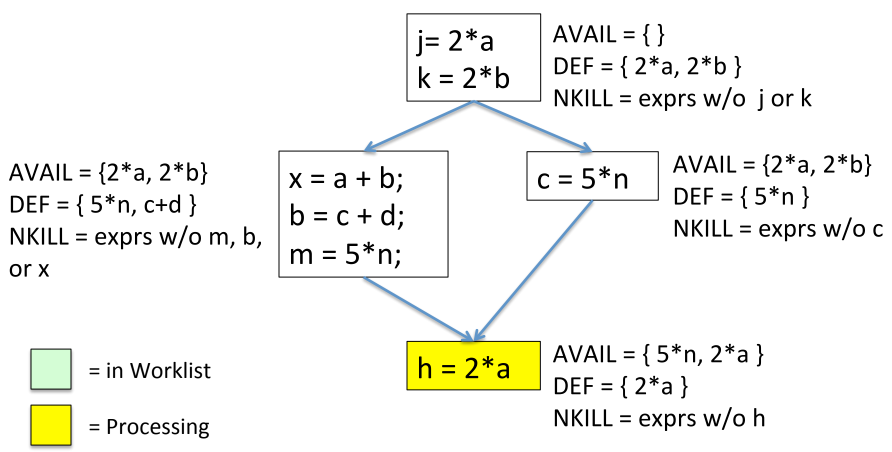
Example: Computing DEF and NKILL

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) $\cup$ (AVAIL(x) $\cap$ NKILL(x)))



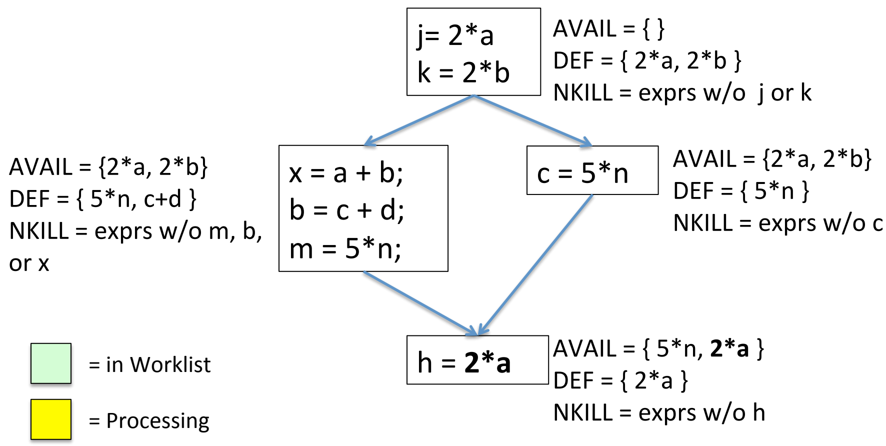
Example: Computing DEF and NKILL

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) $\cup$ (AVAIL(x) $\cap$ NKILL(x)))



Example: Computing DEF and NKILL

AVAIL(b) = $\cap_{x \in preds(b)}$ (DEF(x) ∪ (AVAIL(x) ∩ NKILL(x)))



Dataflow analysis

- Available expressions are an example of a *dataflow analysis* problem
- Many other compiler analyses can be expressed in a similar framework
- Only the first part of the story – once we've discovered facts, we then need to use them to improve code

Characterizing Dataflow Analysis

- All of these algorithms involve sets of facts about each basic block b
  - IN(b) – facts true on entry to b
  - OUT(b) – facts true on exit from b
  - GEN(b) – facts created and not killed in b
  - KILL(b) – facts killed in b
- These are related by the equation

  OUT(b) = GEN(b) ∪ (IN(b) – KILL(b))
  - (Subtracting KILL(b) is equivalent to intersecting NKILL(b))
  - Solve this iteratively for all blocks
  - Sometimes information propagates forward; sometimes backward (reverse in and out)

Example: Live Variable Analysis

- A variable *v* is *live* at point *p* if and only if there is *any* path from *p* to a use of *v* along which *v* is not redefined (i.e., *v* might be used before it is redefined)
- Some uses:
  - Register allocation – registers allocated to live ranges
  - Eliminating useless stores – if variable is not live at store, the stored value will never be used
  - Detecting uses of uninitialized variables – if live at declaration (before initialization), may be used uninitialized.
  - Improve SSA construction – only create phi functions (variable merges) for live variables - coming later …

Liveness Analysis Sets

- For each block b, define
  - use[b] = variable used in b before any def
  - def[b] = variable defined in b before any use
  - in[b] = variables live on entry to b
  - out[b] = variables live on exit from b

Equations for Live Variables

- Given the preceding definitions, we have
  in[b] = use[b] $\cup$ (out[b] – def[b])
  out[b] = $\cup_{s \in succ[b]}$ in[s]
- I.e., live at entry iff this blocks generates liveness (use[b]) or it was live at the exit and this block does not kill liveness (out[b] – def[b]).
- And live at exit iff live at entry to any successor.
- Algorithm
  - Set in[b] = out[b] = $\varnothing$
  - Compute use[b] and def[b] for every block (one time)
  - Update in, out until no change