Northeastern University

CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Credits For Course Material

- Big thank you to UW CSE faculty member, Hal Perkins
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, ...)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, ...)

Agenda

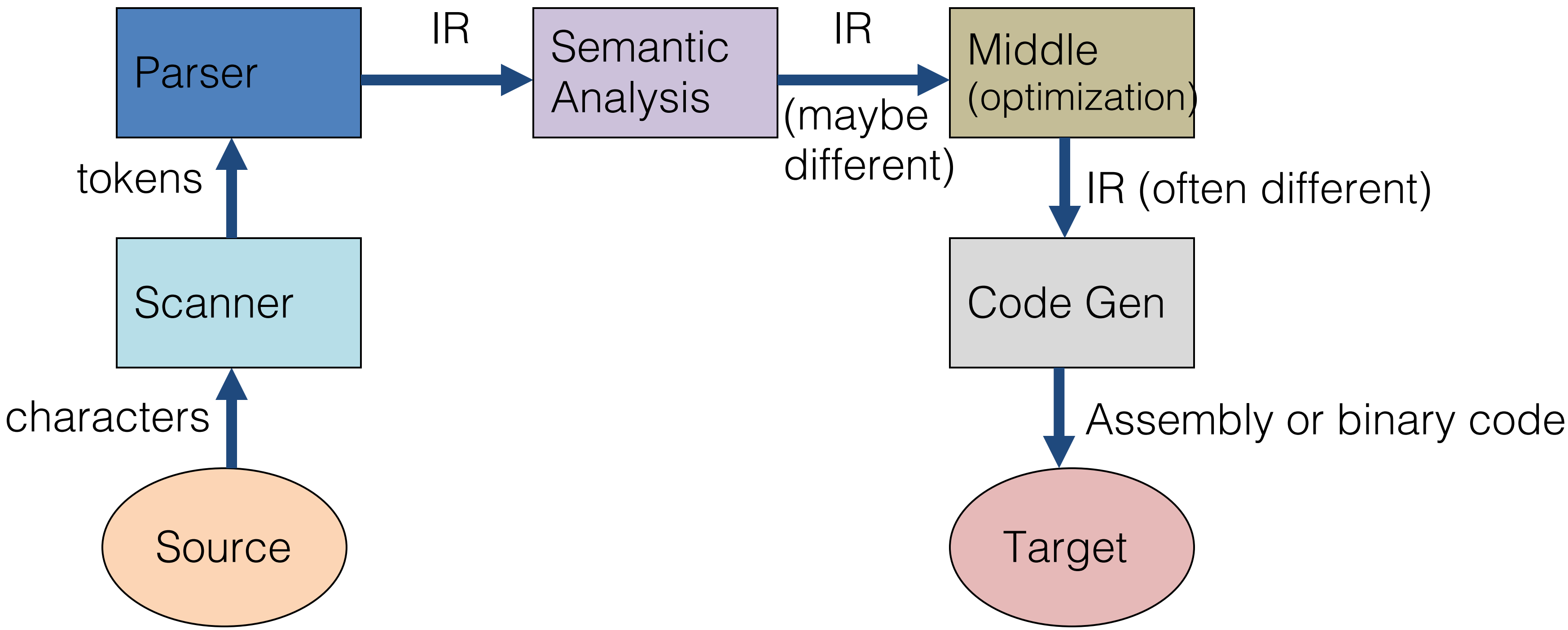
- Static semantics
- Attribute grammars
- Symbol tables
- Types & type checking
- Intermediate representations

Reading:

- Cooper & Torczon – chapter 3 and 5
- The Dragon book, chapters 4 and 6.1, 6.2

Semantics

Review: Compiler Structure



Review: Semantic Analysis

- Main tasks:
  - Extract types and other information from the program
  - Check language rules that go beyond the context-free grammar
  - **Resolve names** – connect declarations and uses
  - **"Understand" the program** – last phase of front end …
  - … so, program is "correct" for hand-off to back end
- Key data structure: Symbol tables
  - For each identifier in the program, record its attributes (kind, type, etc.)
  - Later: assign storage locations (stack frame offsets) for variables, add other annotations

Review: A Sampling of Semantic Checks (0)

- Appearance of a name: id
  - **Check:** id has been declared and is in scope
  - **Compute:** Inferred type of id is its declared type

- Constant: v
  - **Compute:** Inferred type and value are explicit

Review: A Sampling of Semantic Checks (1)

- Binary operator: exp_1 op exp_2
  - **Check:** exp_1 and exp_2 have compatible types
    - Identical, or
    - Well-defined conversion to appropriate types
  - **Compute:** Inferred type is a function of the operator and operand types

Review: A Sampling of Semantic Checks (2)

- Assignment: $exp_1 = exp_2$
  - **Check:** exp_1 is assignable (not a constant or expression)
  - **Check:** exp_1 and exp_2 have (assignment-)compatible types
    - Identical, or
    - exp_2 can be converted to exp_1 (e.g., char to int), or
    - Type of exp_2 is a subclass of type of exp_1 (can be decided at compile time)
  - **Compute:** Inferred type is type of exp_1

Review: A Sampling of Semantic Checks (3)

- Cast: (exp_1) exp_2
  - **Check:** exp_1 is a type
  - **Check:** exp_2 either
    - Has same type as exp_1
    - Can be converted to type exp_1 (e.g., double to int)
    - Downcast: is a superclass of exp_1 (usually requires a runtime check to verify; at compile time we can at least decide if it could be true)
    - Upcast (Trivial): is the same or a subclass of exp_1
  - **Compute:** Inferred type is exp_1

Review: A Sampling of Semantic Checks (4)

- Field reference: exp.f
  - **Check:** exp is a reference type (not value type)
  - **Check:** The class of exp has a field named f
  - **Compute:** Inferred type is declared type of f

Review: A Sampling of Semantic Checks (5)

- Method call: $exp.m(e_1, e_2, \ldots, e_n)$
  - **Check:** exp is a reference type (class instance)
  - **Check:** The class of exp has a method named m
  - **Check:** The method exp.m has n parameters
    - Or, if overloading allowed, at least one version of m exists with n parameters
  - **Check:** Each argument has a type that can be assigned to the associated parameter
    - Same "assignment compatible" check for assignment
    - Overloading: need to find a "best match" among available methods if more than one is compatible – or reject if result is ambiguous (e.g., C++, others)
  - **Compute:** Inferred type is given by method declaration (or could be void)

Review: A Sampling of Semantic Checks (6)

- Return statement: return exp; or: return;
- Check:
  - If the method is not void: The expression can be assigned to a variable with the declared return type of the method – exactly the same test as for assignment statement
  - If the method is void: There is no expression

Attribute Grammars

Attribute Grammars

- A systematic way to think about semantic analysis
- Formalize properties checked and computed during semantic analysis and relate them to grammar productions in the CFG (or AST)
- 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 (abstract) syntax tree
- Examples of attributes
  - Type information
  - Storage location
  - Assignable (e.g., expression vs variable – lvalue vs rvalue in C/C++ terms)
  - Value (for constant expressions)
- 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 the attributes of the Y_i's (bottom up)
- 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 (has implications for evaluation)

Attribute Equations

- For each kind of node we give a set of equations relating attribute values of the node and its neighbors (usually children)
  - Example: $\text{plus.val} = \text{exp}_1\text{.val} + \text{exp}_2\text{.val}$
- Attribution (evaluation) means implicitly finding a solution that satisfies all of the equations in the tree
  - This is an example of a constraint language

Informal Example of Attribute Rules (1)

- Suppose we have the following grammar for a trivial language

  program ::= decl stmt

  decl ::= int id;

  stmt ::= exp = exp ;

  exp ::= id | exp + exp | 1

- What attributes would we create to check types and assignability?

Informal Example of Attribute Rules (2)

- Attributes of nodes
  - env (environment, e.g., symbol table)
    - Synthesized by decl, inherited by stmt
    - Each entry maps a name to its type and kind
  - type (expression type)
    - synthesized
  - kind (variable [var or lvalue] vs value [val or rvalue])
    - synthesized

Attributes for Declarations

decl ::= int id;

decl.env = {id $\longrightarrow$ (int, var)}

Attributes for Program

program ::= decl stmt

stmt.env = decl.env

Attributes for Constants

exp ::= 1

exp.kind = val

exp.type = int

Attributes for Identifier Expressions

exp ::= id

(type, kind) = exp.env.lookup(id)

exp.type = type (i.e., id type)

exp.kind = kind (i.e., id kind)

Attributes for Addition

$exp ::= exp_1 + exp_2$

$exp_1.env = exp.env$

$exp_2.env = exp.env$

error if $exp_1.type \mathrel{!=} exp_2.type$

(or error if not compatible, depending on language rules)

$exp.type = exp_1.type$ (or $exp_2.type$)

$exp.kind = val$

Attribute Rules for Assignment

stmt ::= exp_1 = exp_2;

exp_1.env = stmt.env

exp_2.env = stmt.env

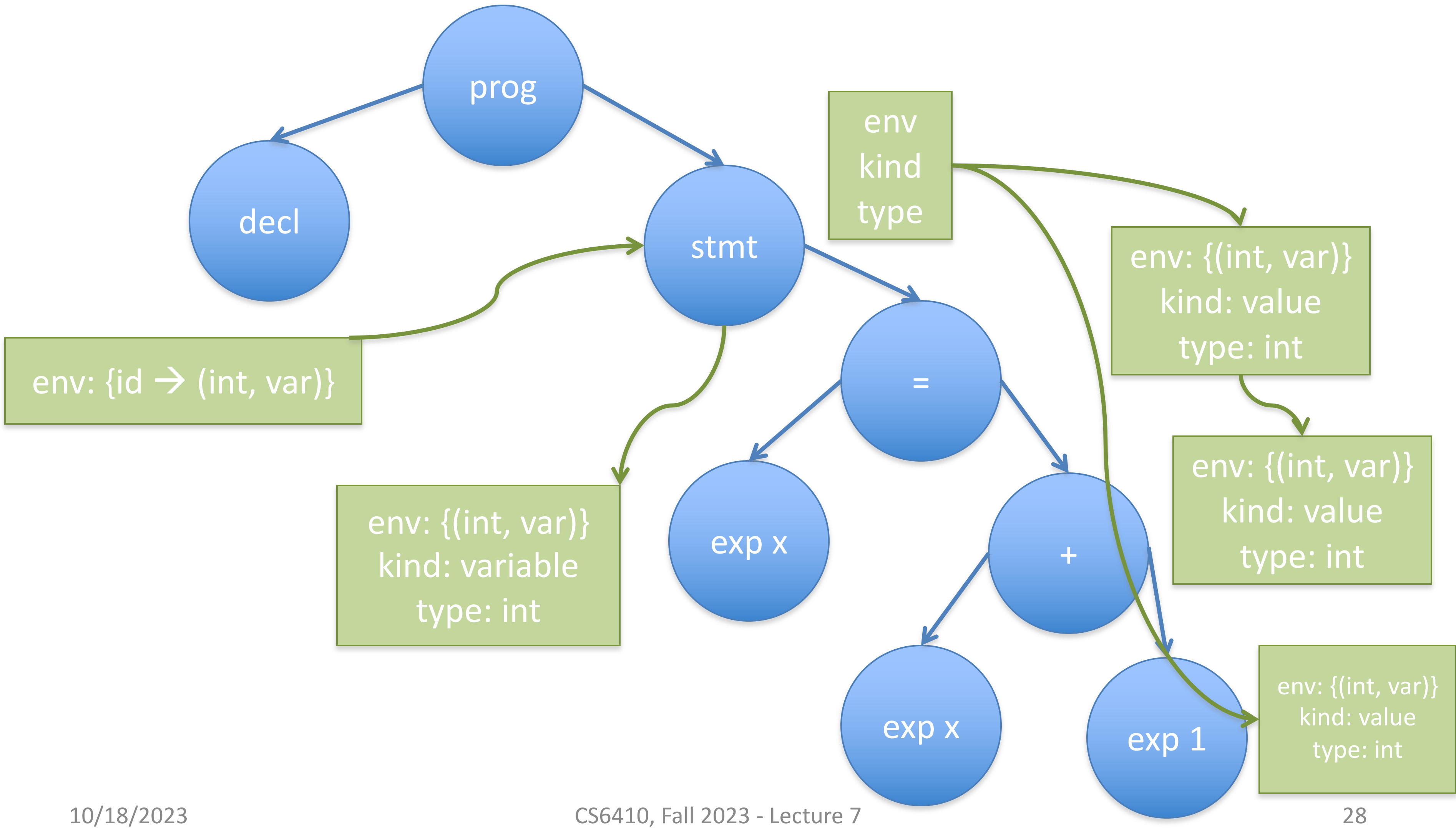
Error if exp_2.type is not assignment compatible with exp_1.type

Error if exp_1.kind is not var (can't be val)

Example

int x; x = x + 1;

Example: int x; x = x + 1;



Extensions

- This can be extended to handle sequences of declarations and statements
  - Sequences of declarations builds up larger environments
  - Each declaration synthesizes a new environment from previous one, plus the new binding
  - Full environment is passed down to statements and expressions

Observations

- These are equational computations
  - Think functional programming, no side effects
- Solver can be automated, provided the attribute equations are non-circular
- But implementation problems
  - Non-local computation
  - Can't afford to literally pass around copies of large, aggregate structures like environments

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 and exploit inheritance. Most statements don't need types, for example, but all expressions do.

Symbol Tables

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 augment) in later compiler phases

Aside: Implementing Symbol Tables

- Big topic in classical (i.e., ancient) compiler courses: implementing a hashed symbol table
- These days: use the collection classes that are provided with the standard language libraries (Java, C#, C++, ML, Haskell, etc.)
  - Then tune & optimize if it really matters
    - In production compilers, it really matters
      - Up to a point…
- Java:
  - Map (HashMap) will handle most cases
  - List (ArrayList) for ordered lists (parameters, etc.)

Symbol Tables for MiniJava

- We'll outline a scheme that does what we need, but feel free to modify/adapt as needed

- Mix of global and local tables

Symbol Tables for MiniJava: Global

- Global – Per Program Information
  - Single global table to map class names to per-class symbol tables
    - Created in a pass over class definitions in AST
    - Used in remaining parts of compiler to check class types and their field/method names and extract information about them

Symbol Tables for MiniJava: Class

- One symbol table for each class
  - One entry per method/field declared in the class
    - Contents: type information, public/private, parameter types (for methods), storage locations (later), etc
- Reached from global table of class names
- In Java, we actually need multiple symbol tables (or more complex symbol table) per class
  - The same identifier can be used for both a method name and a field name in a single class

Symbol Tables for MiniJava: Global/Class

- All global tables persist throughout the compilation
  - And beyond in a real compiler…
    - Symbolic information in Java .class or MSIL files, link-time optimization information in gcc)
    - Debug information in .o and .exe files
    - Some or all information in library files (.a, .so)
    - Type information for garbage collector

Symbol Tables for MiniJava: Methods

- One local symbol table for each method
  - One entry for each local variable or parameter
    - Contents: type info, storage locations (later), etc
  - Needed for project only while compiling the method; can discard when done in a single pass compiler
    - But if type checking and code gen, etc. are done in separate passes, this table needs to persist until we're done with it
      - And beyond: often need type info for runtime debugging, memory management/garbage collection, etc
    - Even for our project, the MiniJava compiler will likely have multiple passes

Beyond MiniJava

- What we aren't dealing with: nested scopes
  - Inner classes
  - Nested scopes in methods – reuse of identifiers in parallel or inner scopes; nested functions (ML, …)
  - Lambdas and function closures
- Basic idea: new symbol table for inner scopes, linked to surrounding scope's table (i.e., stack of symbol tables, top = current innermost scope)
  - Look for identifier in inner scope; if not found look in surrounding scope (recursively)
  - Pop symbol table when we exit a scope
- Also ignoring static fields/methods, accessibility (public, protected, private), package scopes, …

Engineering Issues (1)

- In multipass compilers, inner scope symbol tables need to persist for use in later passes
  - So really can't delete symbol tables on scope exit
  - Retain and add a pointer to the parent scope (effectively a reverse tree of scope symbol tables with root = global table)
    - Keep a pointer to current innermost scope (leaf) and start looking for symbols there

Engineering Issues (2)

- In practice, want to retain O(1) lookup or something close to it
  - Would like to avoid O(depth of scope nesting), although some compilers assume this will be small enough not to matter
  - When it matters, use hash tables with additional information (linked lists of various sorts) to get the scope nesting right
    - Scope entry/exit operators

Error Recovery

- What to do when an undeclared identifier is encountered?
  - Only complain once (Why?)
  - Can forge a symbol table entry for id once you've complained so it will be found in the future
  - Assign the forged entry a type of "unknown"
  - "Unknown" is the type of all malformed expressions and is compatible with all other types
    - Allows you to only complain once! (How?)

"Predefined" Things

- Many languages have some "predefined" items (constants, functions, classes, namespaces, standard libraries, …)
- Include initialization code or declarations to manually create symbol table entries for these when the compiler starts up
  - Rest of compiler generally doesn't need to know the difference between "predeclared" items and ones found in the program
  - Can put "standard prelude" information in a file or data resource and use that to initialize
    - Tradeoffs?

Types and Type Checking

Types

- Classical roles of types in programming languages
  - Run-time safety
  - Compile-time error detection
  - Improved expressiveness (method or operator overloading, for example)
  - Provide information to optimizer
    - In strongly typed languages, allows compiler to make assumptions about possible values

Type Checking Terminology

Static vs. dynamic typing

- **Static:** checking done prior to execution (e.g. compile-time)
- **Dynamic:** checking during execution

Strong vs. weak typing

- **Strong:** guarantees no illegal operations performed
- **Weak:** can't make guarantees

Caveats:

•Hybrids common
•Inconsistent usage common
•"untyped," "typeless" could mean dynamic or weak

	static	dynamic
strong	Java, SML	Scheme, Ruby
weak	C	PERL

Type Systems

- Base Types
  - Fundamental, atomic types
  - Typical examples: int, double, char, bool
- Compound/Constructed Types
  - Built up from other types (recursively)
  - Constructors include records/structs/classes, arrays, pointers, enumerations, functions, modules, …
    - Most language provide a small collection of these

How to Represent Types in a Compiler?

Create a shallow class hierarchy

- Example:

```
   abstract class Type { … }    // or
interface
   class BaseType extends Type { … }
   class ClassType extends Type { … }
```

- Should not need too many of these

Types vs ASTs

- Types nodes are not AST nodes!
- AST = abstract representation of source program (including source program type info)
- Types = abstract representation of type semantics for type checking, inference, etc.
  - Can include information not explicitly represented in the source code, or may describe types in ways more convenient for processing
- Be sure you have a separate “type” class hierarchy in your compiler distinct from the AST

Base Types

- For each base type create exactly one object to represent it (singleton pattern!)
  - Symbol table entries and AST nodes reference these objects to represent entry/node types
  - Usually created at compiler startup
- Useful to create a type "void" object to tag functions that do not return a value
- Also useful to create a type "unknown" object for errors
  - ("void" and "unknown" types reduce the need for special case code in various places in the type checker; don't have to return "null" for "no type" or "not declared" cases)

Compound Types

- Basic idea: use a appropriate “type constructor” object that refers to the component types
  - Limited number of these – correspond directly to type constructors in the language (pointer, array, record/struct/class, function)
  - So a compound type is represented as a graph
- Some examples…

Class Types

- Type for: class Id { fields and methods }

```
class ClassType extends Type {
   Type baseClassType;      // ref to base class
   Map fields;              // type info for
fields
   Map methods;             // type info for
methods
}
```

(MiniJava note: May not want to represent class types exactly like this, depending on how class symbol tables are represented; e.g., the class symbol table(s) might be a sufficient representation of a class type.)

Array Types

- For regular Java this is simple: only possibility is # of dimensions and element type (which can be another array type or anything else)

```
class ArrayType extends Type {
   int nDims;
   Type elementType;
}
```

Methods/Functions

- Type of a method is its result type, plus an ordered list of parameter types

```
class MethodType extends Type {
    Type resultType;          // type or
"void"
    List parameterTypes;
}
```

- Sometimes called the method “signature”

Type Equivalance

- For base types this is simple: types are the same if they are identical
    - Can use pointer comparison in the type checker if you have a singleton object for each base type
  - Normally there are well defined rules for coercions between arithmetic types
    - Compiler inserts these automatically where required by the language spec or when written explicitly by programmer (casts) – often involves inserting cast or conversion nodes in AST

Type Equivalence for Compound Types

- Two basic strategies
  - *Structural equivalence*: two types are the same if they are the same kind of type and their component types are equivalent, recursively
  - *Name equivalence*: two types are the same only if they have the same name, even if their structures match
- Different language design philosophies
  - e.g., are Complex and Point the same?
  - e.g., are Point (Cartesian) and Point (Polar) the same?

Structural Equivalence

- Structural equivalence says two types are equal iff they have same structure
  - Atomic types are tautologically the same structure and equal if they are the same type
  - For type constructors: equal if the same constructor and, recursively, type (constructor) components are equal
- Ex: atomic types, array types, ML record types
- Implement with recursive implementation of equals, or by canonicalization of types when types created, then use pointer/ref. equality

Name Equivalence

- Name equivalence says that two types are equal iff they came from the same textual occurrence of a type constructor
  - Ex: class types, C struct types (struct tag name), datatypes in ML
  - special case: type synonyms (e.g. typedef in C) do not define new types
- Implement with pointer equality assuming appropriate representation of type info

Type Equivalence and Inheritance

- Suppose we have

```
class Base { … }
class Derived extends Base { … }
```

- A variable declared with type Base has a *compile-time type* or *static type* of Base
- During execution, that variable may refer to an object of class Base or any of its subclasses like Derived (or can be null), often called the the *runtime type* or *dynamic type*
  - Since subclass is guaranteed to have all fields/methods of base class, type checker only needs to deal with declared compile-time types of variables and, in fact, can't track all possible runtime types

Type Casts

- In most languages, one can explicitly cast an object of one type to another
  - Sometimes cast means a conversion (e.g., casts between numeric types)
  - Sometimes cast means a change of static type without doing any computation (casts between pointer types or pointer and numeric types in C)
  - For objects can be a upcast (free and always safe) or downcast (requires runtime check to be safe)

Type Conversions and Coercions

- In full Java, we can explicitly convert a value of type double to one of type int
  - can represent as unary operator
  - typecheck, codegen normally
- In full Java, can implicitly coerce an value of type int to one of type double
  - compiler must insert unary conversion operators, based on result of type checking

C and Java: type casts

- In C/C++: safety/correctness of casts not checked
  - Allows writing low-level code that's type-unsafe
  - C++ has more elaborate casts, and at least one of them does imply runtime checks
- In Java: downcasts from superclass to subclass need runtime check to preserve type safety
    - static typechecker allows the cast
    - codegen introduces runtime check
      - (same code needed to handle "instanceof")
    - Java's main need for dynamic type checking

Various Notions of Type Compatibility

- There are usually several relations on types that we need to analyze in a compiler:
  - “is the same as”
  - “is assignable to”
  - “is same or a subclass of”
  - “is convertible to”
- Exact meanings and checks needed depend on the language specifications
- Be sure to check for the right one(s)

Useful Compiler Functions

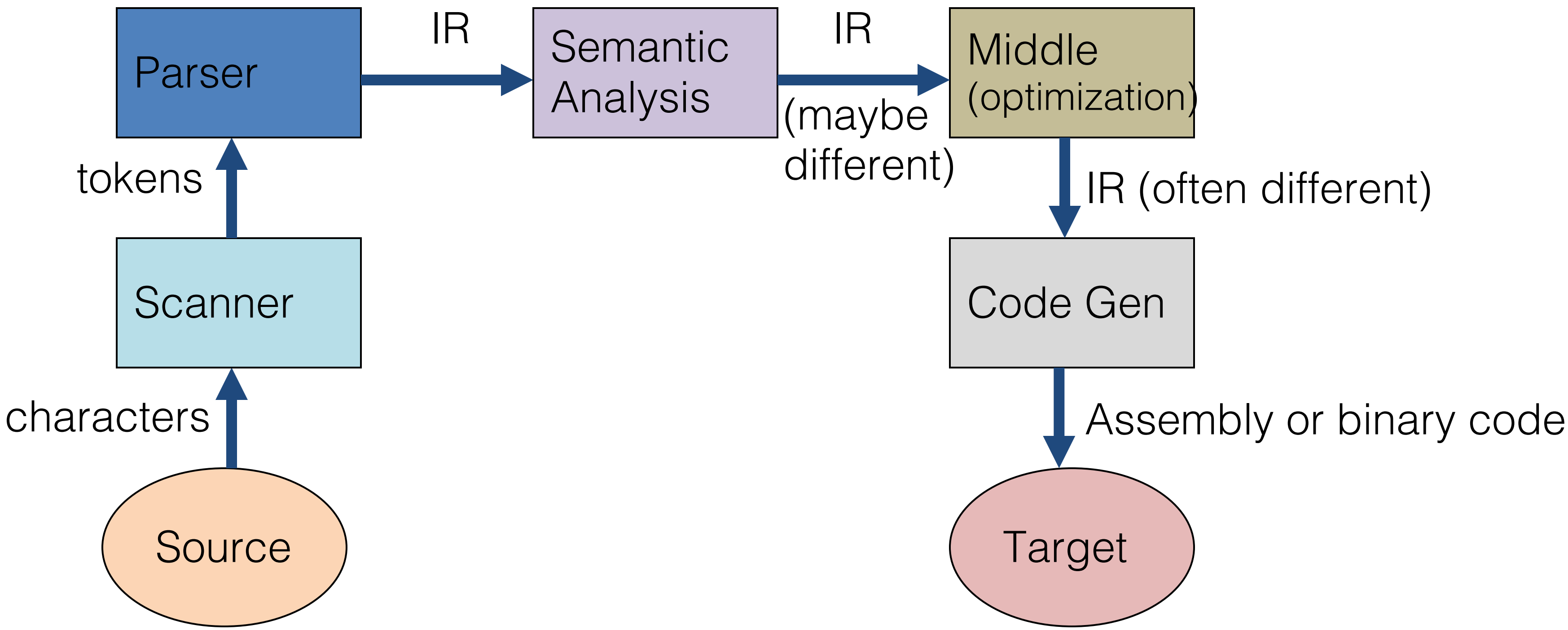
- Create a handful of methods to decide different kinds of type compatibility:
  - Types are identical
  - Type t_1 is assignment compatible with t_2
  - Parameter list is compatible with types of expressions in the method call
- Usual modularity reasons: isolates these decisions in one place and hides the actual type representation from the rest of the compiler
- Probably belongs in the same package with the type representation classes

Implementing Type Checking for MiniJava

- Create multiple visitors for the AST
- First pass/passes: gather information
  - Collect global type information for classes
  - Could do this in one pass, or might want to do one pass to collect class information, then a second one to collect per-class information about fields, methods – you decide
- Next set of passes: go through method bodies to check types, other semantic constraints

Intermediate Representations

Review: Compiler Structure



Intermediate Representations

- In most compilers:
  - The parser builds an intermediate representation (IR, typically an Abstract Syntax Tree)
  - Rest of the compiler transforms the IR to optimize it
  - Typically will transform initial IR to one or more different IRs along the way
  - IR eventually translate to final target code

IR Design

- Decisions affect speed and efficiency of the rest of the compiler
  - General rule: compile time is important, but performance of generated code often more important
  - Typical case for production code: compile a few times, run many times
    - Although the reverse is true during development
  - So make choices that improve compile time as long as they don't compromise the result

IR Design

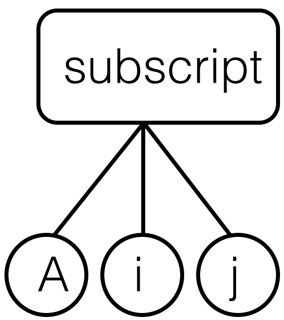
- Desirable properties
  - Easy to generate
  - Easy to manipulate
  - Expressive
  - Appropriate level of abstraction
- Different tradeoffs depending on compiler goals
- Different tradeoffs in different parts of the same compiler
  - So often different IRs in different parts

IR Design Taxonomy

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

Examples: Array Reference

source: A[i,j]



t1 ← A[i,j]

```
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
```

Levels of Abstraction

- Key design decision: how much detail to expose
  - Affects possibility and profitability of various optimizations
    - Depends on compiler phase: some semantic analysis & optimizations are easier with high-level IRs close to the source code. Low-level usually preferred for other optimizations, register allocation, code generation, etc.
  - Structural (graphical) IRs are typically fairly high-level
  - Linear IRs are typically low-level

Graphical IRs

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

Concrete Syntax Trees

- The full grammar is needed to guide the parser, but contains many extraneous details
  - Chain productions
  - Rules that control precedence and associativity
- Typically the full syntax tree (parse tree) does not need to be used explicitly, but sometimes we want it (structured source code editors or transformations, …)

Example

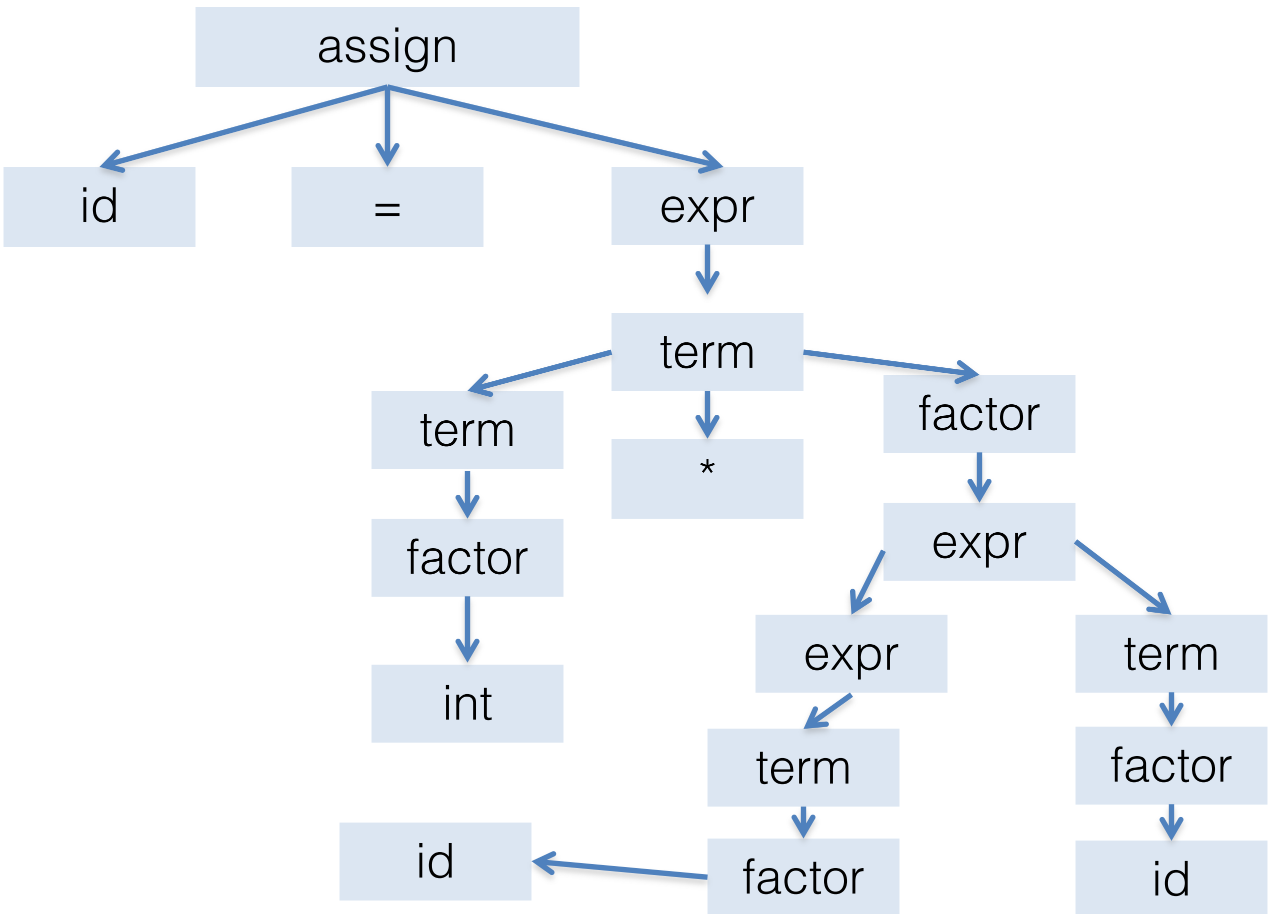
- Concrete syntax for x = 2*(n+m)

assign ::= *id* = *expr* ;
expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | *id* | (*expr*)

assign ::= *id* = *expr* ;
expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | *id* | (*expr*)

Example

- Concrete syntax for x = 2*(n+m)



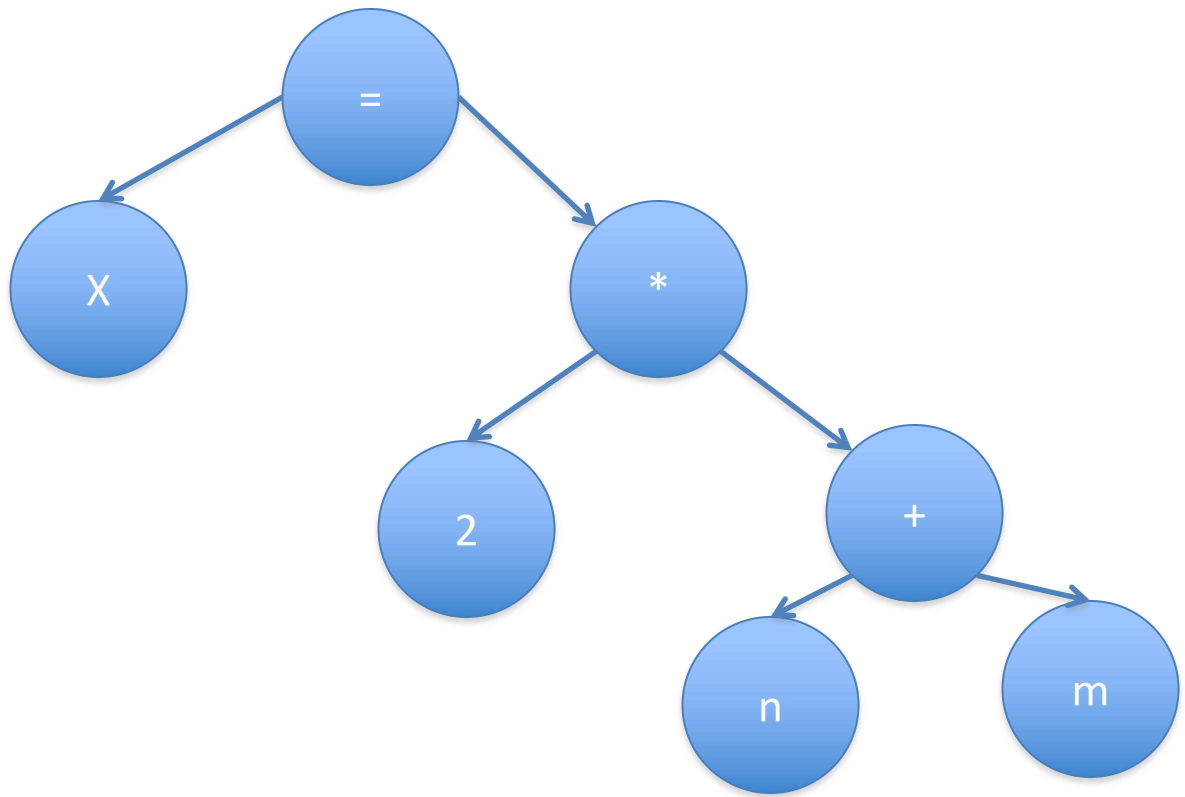
Abstract Syntax Trees

- Common output from parser:
  - Used for static semantics (type checking, etc)
  - Sometimes used high-level optimizations
- Focus on essential structural information
- Can be represented:
  - Explicitly as a tree or
  - In a linear form
- Example: LISP/Scheme S-expressions are essentially ASTs

assign ::= *id* = *expr* ;
expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | *id* | (*expr*)

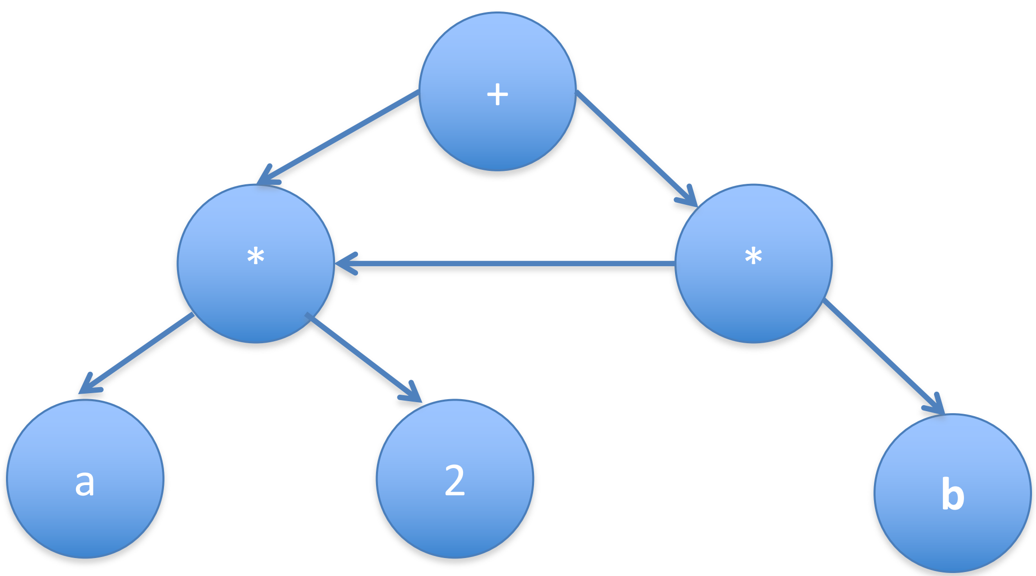
Example

- Abstract syntax for x = 2*(n+m)



DAGs (Directed Acyclic Graphs)

- Variation on ASTs with shared substructures
- Pro: saves space, exposes redundant sub-expressions
- Con: less flexibility if part needs to be changed



Linear IRs

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

```
t1 ← 2
t2 ← b
t3 ← t1 *
t2
t4 ← a
t5 ← t4 –
t3
```

- Fairly compact
- Compiler can control reuse of names – clever choice can reveal optimizations
- ILOC & similar code

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

- Each instruction consumes top of stack & pushes result
- Very compact
- Easy to create and interpret
- Java bytecode, MSIL

Abstraction Levels in Linear IR

- Linear IRs can be:
  - High-level abstraction (close to the source language)
  - Medium-level abstraction
  - Very low-level abstraction
- Examples: Linear IRs for C array reference a[i][j+2]
  - High-level: t1 $\leftarrow$ a[i,j+2]

More IRs for a[i][j+2]

- Medium-level

t1 ← j + 2
t2 ← i * 20
t3 ← t1 + t2
t4 ← 4 * t3
t5 ← addr a
t6 ← t5 + t4
t7 ← *t6

- Low-level

r1 ← [fp-4]
r2 ← r1 + 2
r3 ← [fp-8]
r4 ← r3 * 20
r5 ← r4 + r2
r6 ← 4 * r5
r7 ← fp – 216
f1 ← [r7+r6]

Abstraction Level Tradeoffs

- High-level abstraction:
  - Good for some source-level optimizations and semantic checking
  - Can’t optimize things that are hidden – like address arithmetic for array subscripting
- Medium-level abstraction:
  - More details, but keeps more higher-level semantic information – great for machine-independent optimizations
  - Many (all?) optimizing compilers work at this level
- Low-level abstraction:
  - Need for good code generation and resource utilization in back end
  - Loses semantic knowledge (e.g., variables, data aggregates, source relationships are usually missing)
- Many compilers use all 3 in different phases

Three-Address Code (TAC)

- Usual form: $x \leftarrow y \text{ op } z$
  - One operator
  - Maximum of 3 names
  - (Copes with: nullary $x \leftarrow y$ and unary $x \leftarrow \text{op } y$)

- Eg: $x = 2 * (m + n)$ becomes

  $t1 \leftarrow m + n;\quad t2 \leftarrow 2 * t1;\quad x \leftarrow t2$
  - You may prefer: add t1, m, n; mul t2, 2, t1; mov x, t2
- "Expression temps":
  - Invent as many new temp names as needed
  - Don't correspond to any user variables; de-anonymize expressions
- Store in a quad(ruple)
  - <lhs, rhs1, op, rhs2>

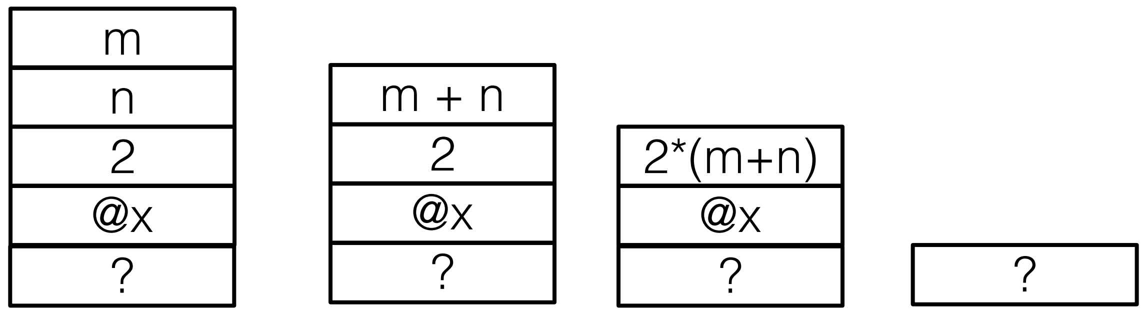
Three Address Code

- Advantages
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange
- Various representations
  - Quadruples, triples, SSA (Static Single Assignment)
  - We will see much more of this…

Stack Machine Code Example

Hypothetical code for x = 2 * (m + n)

```
pushaddr    x
pushconst   2
pushval     n
pushval     m
add
mult
store
```



Compact: common opcodes just 1 byte wide; instructions have 0 or 1 operand

Stack Machine Code

- Originally used for stack-based computers (famous example: B5000, ~1961)
- Now also used for virtual machines:
  - UCSD Pascal – pcode
  - Forth
  - Java bytecode in a .class files (generated by Java compiler)
  - MSIL in a .dll or .exe assembly (generated by C#/F#/VB compiler)
- Advantages
  - Compact; mostly 0-address opcodes (fast download over network)
  - Easy to generate; easy to write a FrontEnd compiler, leaving the 'heavy lifting' and optimizations to the JIT
  - Simple to interpret or compile to machine code
- Disadvantages
  - Inconvenient/difficult to optimize directly
  - Does not match up with modern chip architectures

Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- Most common example: control-flow graph (CFG)

Control Flow Graph (CFG)

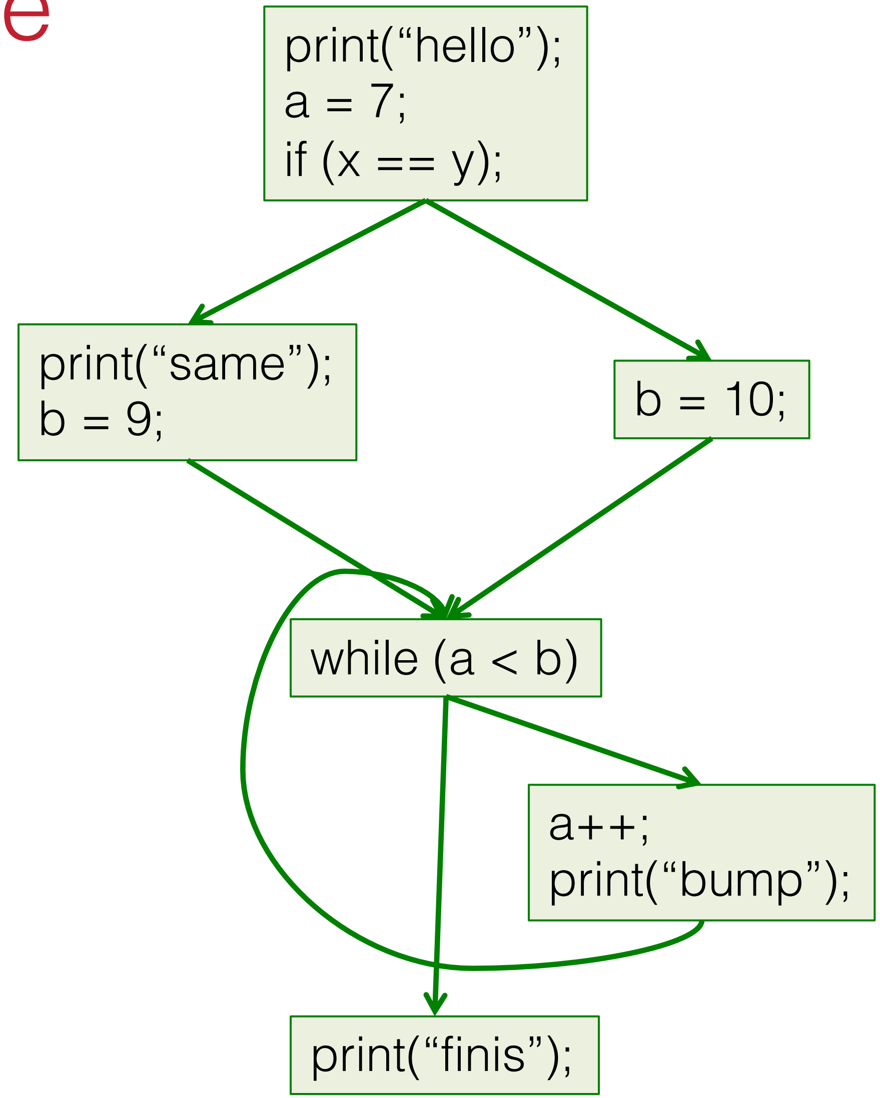
- Nodes: basic blocks
- Edges: possible flow of control from one block to another, (i.e., possible execution orderings)
  - Edge from A to B if B could execute immediately after A in some possible execution
- Required for much of the analysis done during optimization phases

Basic Blocks

- Fundamental concept in analysis/optimization
- A *basic block* is:
  - A sequence of code
  - One entry, one exit
  - Always executes as a single unit ("straight-line code") – so it can be treated as an indivisible block
    - We'll ignore exceptions, at least for now
- Usually represented as some sort of a list although Trees/DAGs are possible

CFG Example

```
print(“hello”);
a=7;
if (x == y) {
  print(“same”);
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print(“bump”);
}
print(“finis”);
```



Basic Blocks: Start with Tuples

```
1 i = 1
2 j = 1
3 t1 = 10 * i
4 t2 = t1 + j
5 t3 = 8 * t2
6 t4 = t3 - 88
7 a[t4] = 0
8 j = j + 1
9 if j <= 10 goto #3
10 i = i + 1
11 if i <= 10 goto #2
12 i = 1
13 t5 = i - 1
14 t6 = 88 * t5
15 a[t6] = 1
16 i = i + 1
17 if i <= 10 goto #13
```

Typical "tuple stew" - IR generated by traversing an AST

Partition into Basic Blocks:

- Sequence of consecutive instructions
- No jumps into the middle of a BB
- No jumps out of the middles of a BB
- "I've started, so I'll finish"
- (Ignore exceptions)

Basic Blocks: Leaders

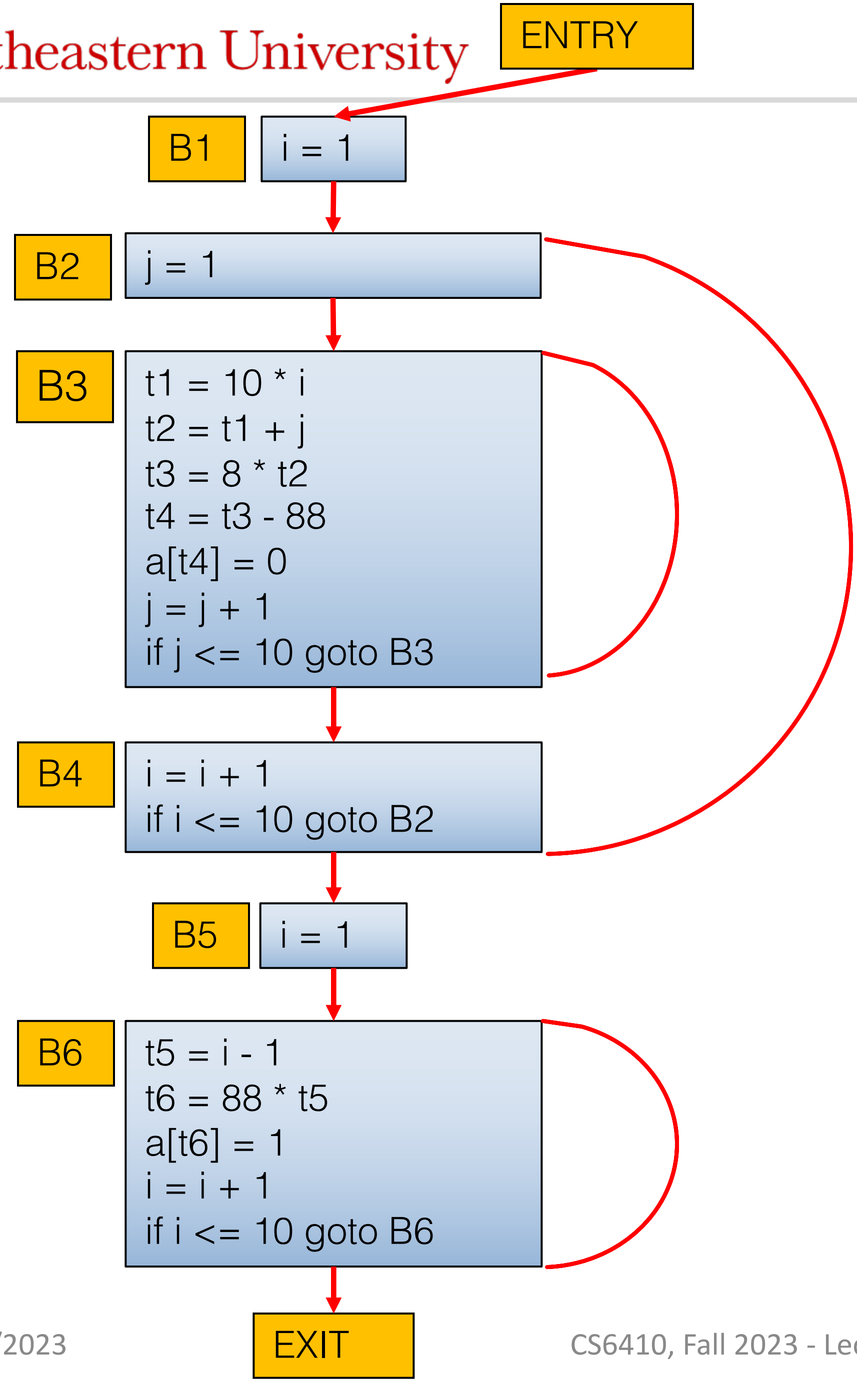
```
1 i = 1                    10 i = i + 1
2 j = 1                    11 if i <= 10 goto #2
3 t1 = 10 * i              12 i = 1
4 t2 = t1 + j              13 t5 = i - 1
5 t3 = 8 * t2              14 t6 = 88 * t5
6 t4 = t3 - 88             15 a[t6] = 1
7 a[t4] = 0                16 i = i + 1
8 j = j + 1                17 if i <= 10 goto #13
9 if j <= 10 goto #3
```

Identify Leaders (first instruction in a basic block):

- First instruction is a leader
- Any target of a branch/jump/goto
- Any instruction immediately after a branch/jump/goto

Leaders in red. Why is each leader a leader?

Basic Blocks: Flowgraph



Control Flow Graph ("CFG", again!)

- 3 loops total
- 2 of the loops are nested

Most of the executions likely spent in loop bodies; that's where to focus efforts at optimization

Identifying Basic Blocks: Recap

- Perform linear scan of instruction stream
- A basic blocks begins at each instruction that is:
  - The beginning of a method
  - The target of a branch
  - Immediately follows a branch or return

Dependency Graphs

- Often used in conjunction with another IR
- Data dependency: edges between nodes that reference common data
- Examples
  - RAW – read after write: Block A defines x then B reads it
  - WAR – "anti-dependence": Block A reads x then B writes it
  - WAW: Blocks A and B both write x – order of blocks must reflect original program semantics
- These restrict reorderings the compiler can do

What IR to Use?

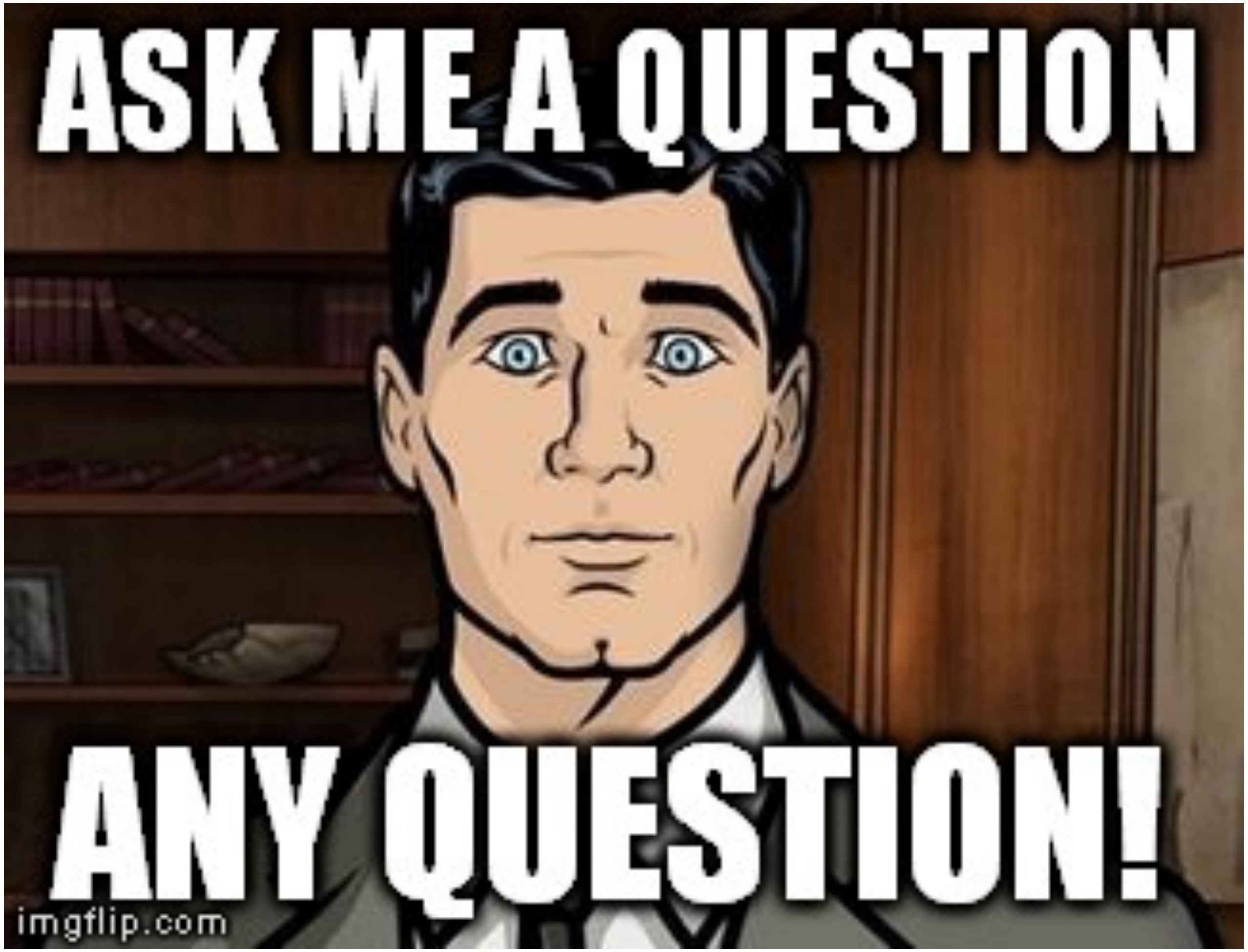
- Common choice: all(!)
  - AST used in early stages of the compiler
    - Closer to source code
    - Good for semantic analysis
    - Facilitates some higher-level optimizations
  - Lower to linear IR for optimization and codegen
    - Closer to machine code
    - Use to build control-flow graph
    - Exposes machine-related optimizations
  - Hybrid (graph + linear IR = CFG) for dataflow & opt

Coming Attractions

- To get a running compiler we need:
  - Execution model for language constructs
  - x86-64 assembly language for compiler writers
  - Code generation and runtime bootstrap details

- We'll also spend considerable time on compiler optimization
  - Intermediate reps., graphs, SSA, dataflow
  - Optimization analysis and transformations

- Immediate problem is to keep lectures from getting too far ahead of the project - maybe hold off on runtime details?
  - Thoughts? Suggestions? Opinions?



[Meme credit: imgflip.com]