Compilers

by
Dr. Marwa Yusuf

Lecture 1
Mon. 13-2-2023

Chapter 1
Introduction

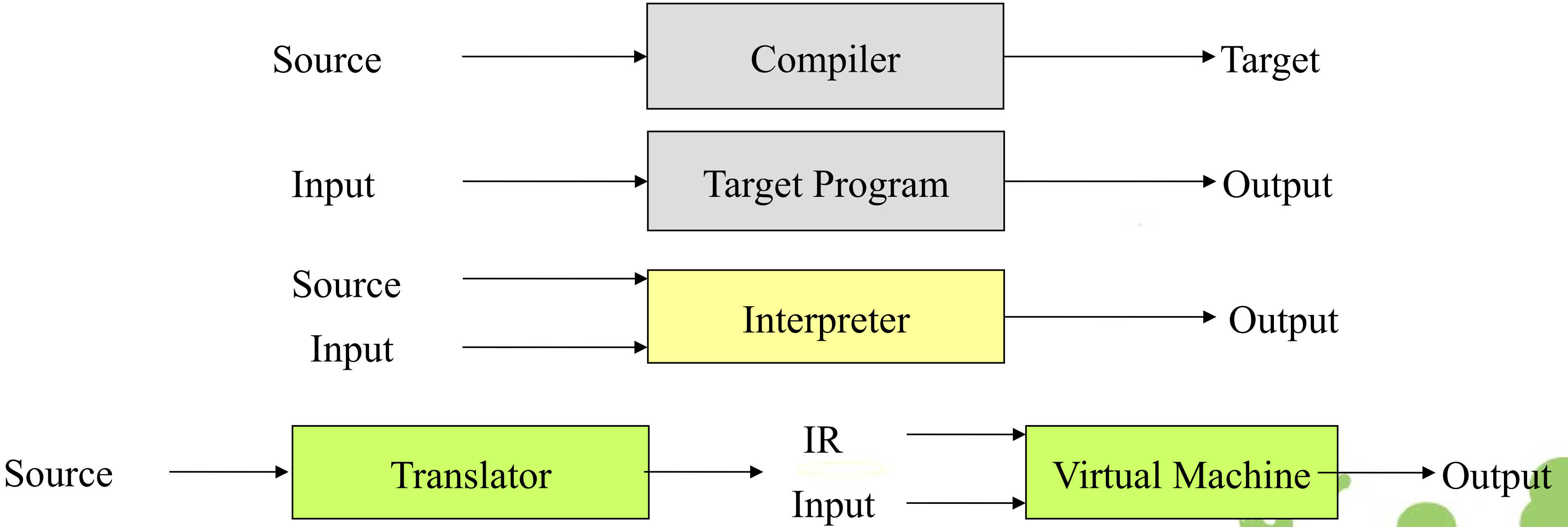
Course Info.

- Grading
  - Final: 90
  - Midterm: 30
  - Project: 15
  - Lab: 9
  - Quizzes: 6
- MS Team
  - Code: **tdsd8v3**

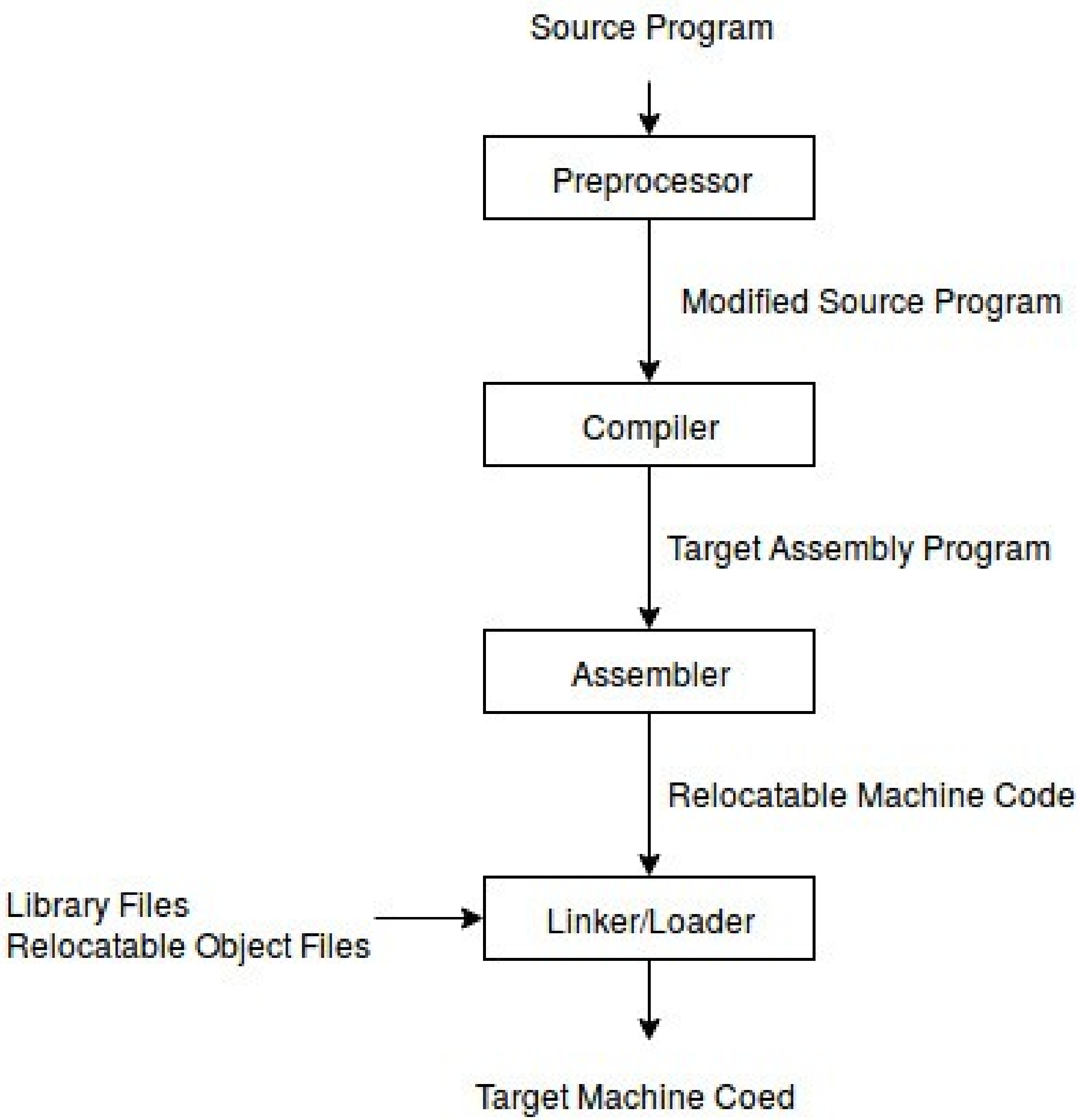
Course Description

- [Compiler Course Specs]

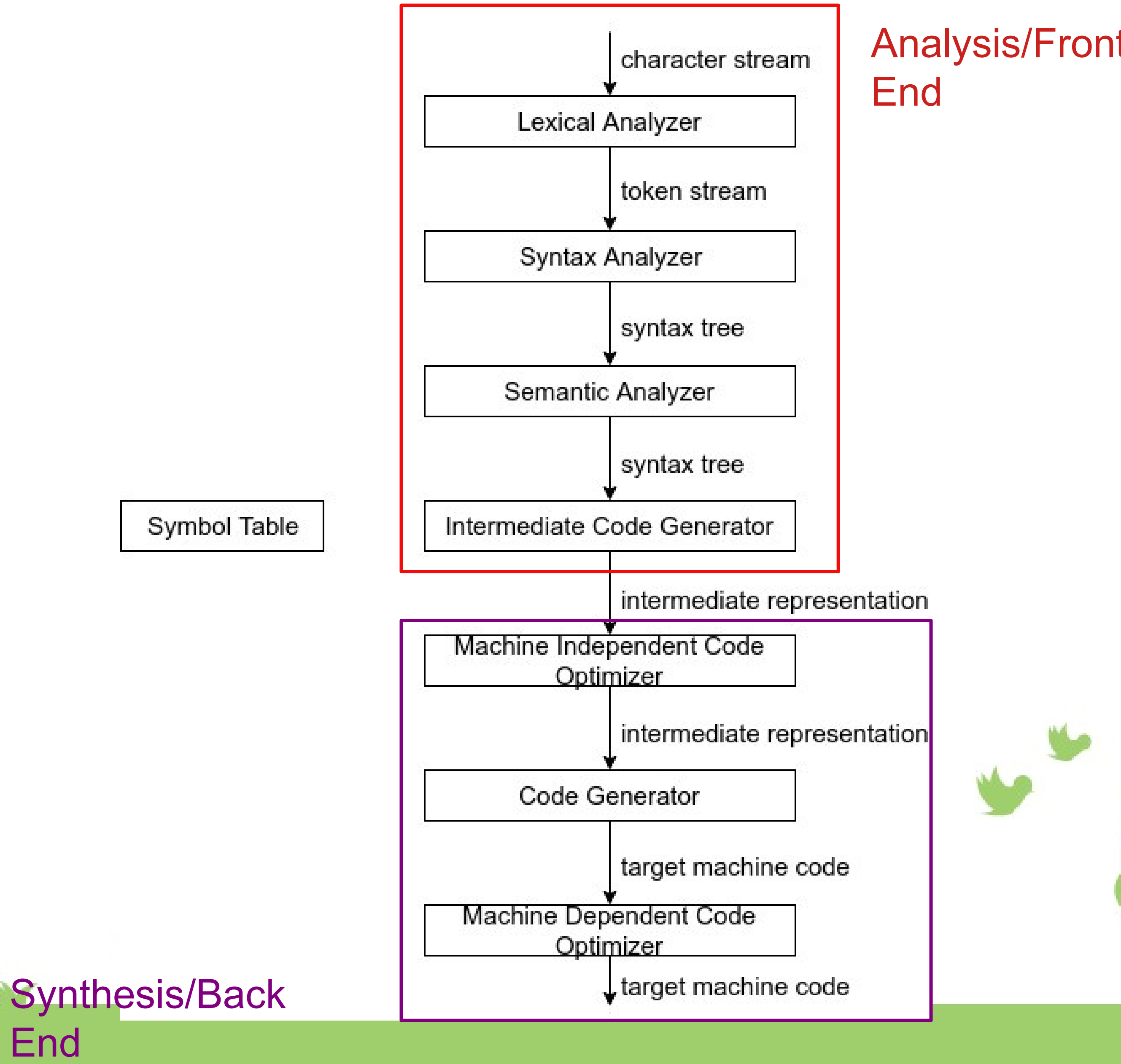
Language Processors



From Source to Execution



The structure of a Compiler



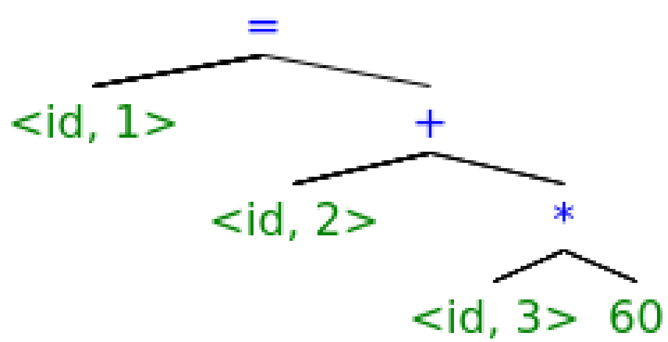
Lexical Analysis

- Called scanning
- in: character stream, out: lexemes, tokens
  - <token-name, attribute-value>
  - attribute-value: points to an entry in symbol table
- Ex: `position = initial + rate * 60`
  - <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

position	...
initial	...
rate	...

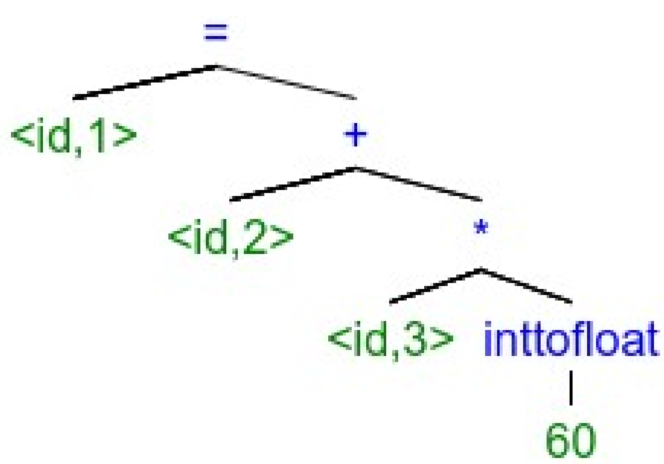
Syntax Analysis

- Called parsing
- in: tokens, out: syntax tree
- Ex: <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>



Semantic Analysis

- in: syntax tree & symbol table
- Check that it conforms with language def.
- Save type info.
- Type checking (coercions when needed)



Intermediate Code Generation

- Generate IR (syntax tree is a form of IR).
- IR: easy to produce, easy to translate.
- Ex: three-address-code

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimization

- Enhance IR to get better machine code later.
  - Execution time, shorter code, less power
- Ex:

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation

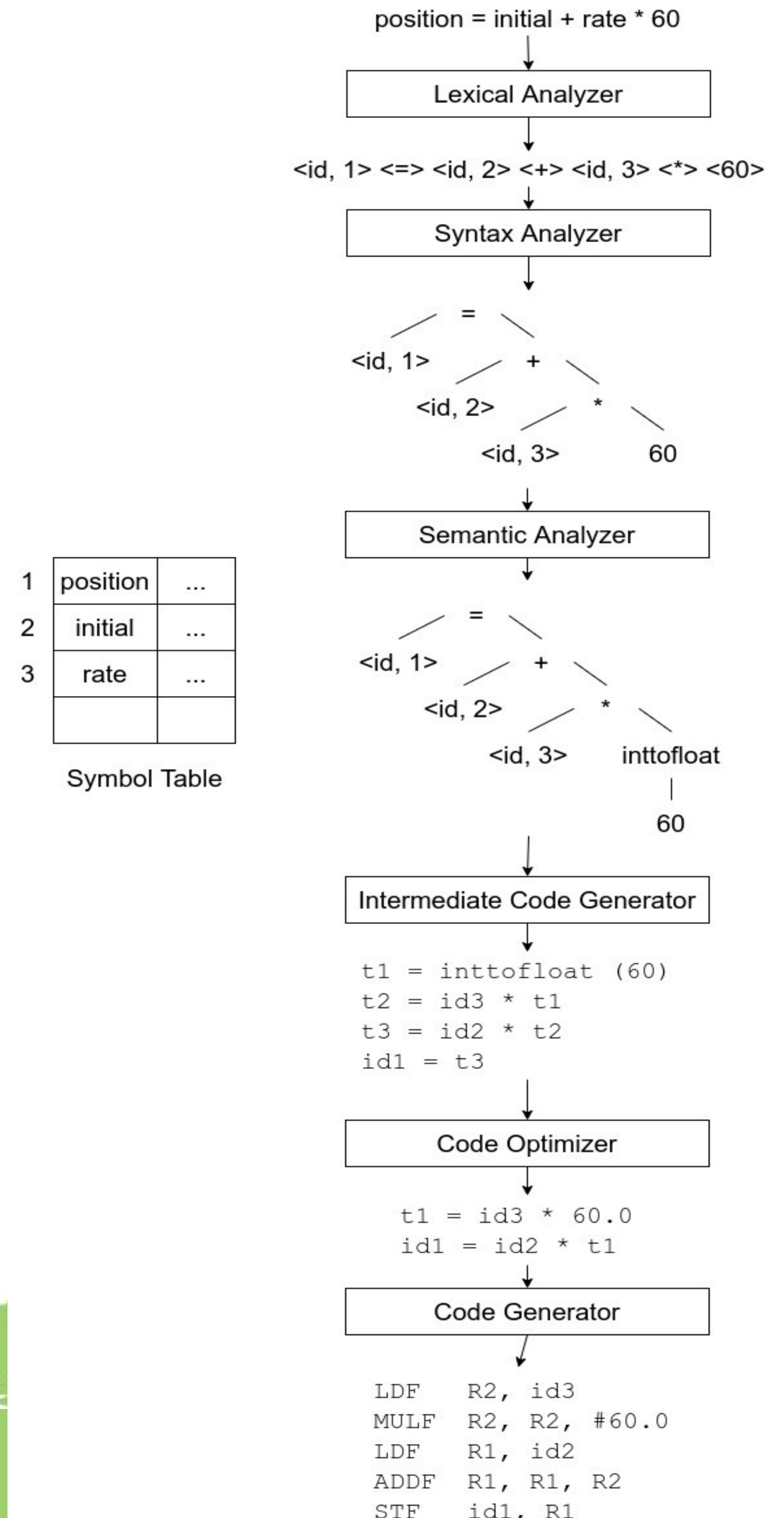
- in: IR, out: machine code
- Register allocation, storage allocation (may be done in IR generation)
- Ex:

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

Symbol Table Management

- Record variable names and their attributes:
  - name, type, scope
  - Procedures: number & types of arguments, passing method, return type
- Should be efficient.


position = initial + rate * 60
Lexical Analyzer
<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>
Syntax Analyzer
=
<id, 1>
+
<id, 2>
*
<id, 3>
60
Semantic Analyzer
1 position ...
2 initial ...
3 rate ...
Symbol Table
=
<id, 1>
+
<id, 2>
*
<id, 3>
inttofloat
60
Intermediate Code Generator
t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 * t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 * t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

Grouping phases into passes

- Front end: lexical, syntax, semantic and IR generation
- Optional optimization pass
- Back end: code generation
- Some compiler collections: several front ends & several back ends (ex: LLVM)

Compiler Construction Tools

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Code-generator generators
- Data-flow analysis engines
- Compiler construction toolkits

Programming Languages Evolution

- 1st generation: machine language (1940's), simple instructions.
- 2nd generation: assembly (early 1950's), first just mnemonics, then macro instructions.
- 3rd generation: high level (latter half of 1950's) (Fortran, Cobol, Lisp, C, C++, C#, Java)
- 4th generation: SQL, for specific applications.
- 5th generation: Prolog, logic and constraint-based.

- Imperative vs. declarative (Haskell)
- Von Neumann languages.
- OOP languages.
- Scripted languages: interpreted, gluing

Programming Languages Evolution (cont.)

- Affect compiler design.
- Architecture development affects also.
- Compilers used to evaluate new arch.
- Compiler writing is challenging.
- Generated code must be correct, and preferably efficient (undecidable).

The science of Building a Compiler

- Abstract the problem into models.
- Examples:
  - Finite state machines
  - Regular expressions
  - Context-free grammars
  - Trees
- Optimizations:
  - Correct, improving many programs, reasonable compilation time and manageable engineering effort

Applications of Compiler Technology

- Implementation of high level languages:
  - While programming in lower level language may provide more control, more advanced compilers make it easy to use higher level languages.
  - **Ex: register** in C
  - Java: garbage collection, type-checking, range checks, interpreter, JIT
- Optimizations for computer architectures:
  - Parallelism:VLIW, instructions on a data vector, multiprocessors.
  - Memory hierarchy: registers, layout of data and order of access affects cache performance.

Applications of Compiler Technology (cont.)

- Design of New Computer Architectures:
  - Currently, compilers developed during processor-design stage, and used (on simulators) to evaluate new arch.
  - RISC vs CISC
  - Specialized architectures: vector machines, VLIW, SIMD, ...
- Program Translations:
  - Binary translation: from a machine to another, backward compatibility.
  - H/w synthesis (VHDL).
  - Database query interpreters.
  - Compiled simulation
- Software productivity tools
  - Data flow analysis, type checking, bounds checking, memory management tools

Programming Language Basics

- Dynamic vs. static: scope, memory location (class data)
- Environments (mapping from names to variables/locations) and states (mapping from variables to values).
  - Ex: global vs local variables.

```
int x;   //global x
void f(…) {
    int x;      //local x
    x = 3;
}
y = x + 1;
```

  - Static vs. dynamic binding in both cases:
    - Most are dynamic, but some are static like globals in C and declared constants.

Programming Language Basics (cont.)

- Static scope and block structure.
  - Explicit access control (public, private, protected)
  - Block scope. Ex:

```
main() {
    int a = 1;                B1
    int b = 1;
    {
        int b = 2;            B2
        {
            int a = 3;        B3
            cout << a << b;
        }
        {
            int b = 4;        B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Declaration	Scope
int a = 1	B1 - B3
int b = 1	B1 - B2
int b = 2	B2 - B4
int a = 3	B3
int b = 4	B4

Programming Language Basics (cont.)

- Explicit Access Control:
  - Using private, protected and public.
- Dynamic scope:
  - Macro expansion in C

```
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("%d\n", a); }
void c() { printf(%d\n", a); }
void main() { b(); c(); }
```

- Methods in java.
  - Superclass `C` and subclass `D` and method `m` declared in both: `x.m()`

Programming Language Basics (cont.)

- Parameter passing mechanisms:
  - by value, by reference, by name.
- Aliasing:
  - Ex: `q(x, y)` called as `q(a, a)`
  - Affects possibility of optimization

Thanks!
And Let It Begin