CSCE 314
Programming Languages

A Tour of Language Implementation

Dr. Hyunyoung Lee

Programming Language Characteristics

Different approaches to

- describe computations, instruct computing devices

  E.g., Imperative, declarative, functional

- communicate ideas between humans

  E.g., Procedural, object-oriented, domain-specific languages

Programming language specification: meaning (semantics) of all sentences (program syntax) of the language should be unambiguously specified

Programming Language Expressiveness

Different levels of abstraction

More abstract

Haskell, Prolog	sum[1..100]
Scheme, Java	mynum.add(5)
C	i++;
Assembly language	iadd
Machine language	10111001010110

Evolution of Languages

- 1940's: connecting wires to represent 0's and 1's
- 1950's: assemblers, FORTRAN, COBOL, LISP
- 1960's: ALGOL, BCPL (→ B → C), SIMULA
- 1970's: Prolog, FP, ML, Miranda
- 1980's: Eiffel, C++
- 1990's: Haskell, Java, Python
- 2000's: D, C#, Spec#, F#, X10, Scala, Ruby, . . .
- 2010's: Agda, Coq
- . . .

Evolution has been and is toward higher level of abstraction

Defining a Programming Language

- Syntax: Defines the set of valid programs

  Usually defined with the help of grammars and other conditions

  if-statement ::= if *cond-expr* then *stmt* else *stmt*
  | if *cond-expr* then *stmt*

  cond-expr ::= . . .

  stmt ::= . . .

- Semantics: Defines the meaning of programs

  Defined, e.g., as the effect of individual language constructs to the values of program variables

  if *cond* then *true-part* else *false-part*

  If *cond* evaluates to true, the meaning is that of *true-part*; if cond evaluates to false, the meaning is that of *false-part*

Implementing a Programming Language

- Task is to undo abstraction. From the source:

```
int i;
i = 2;
i = i + 7;
```

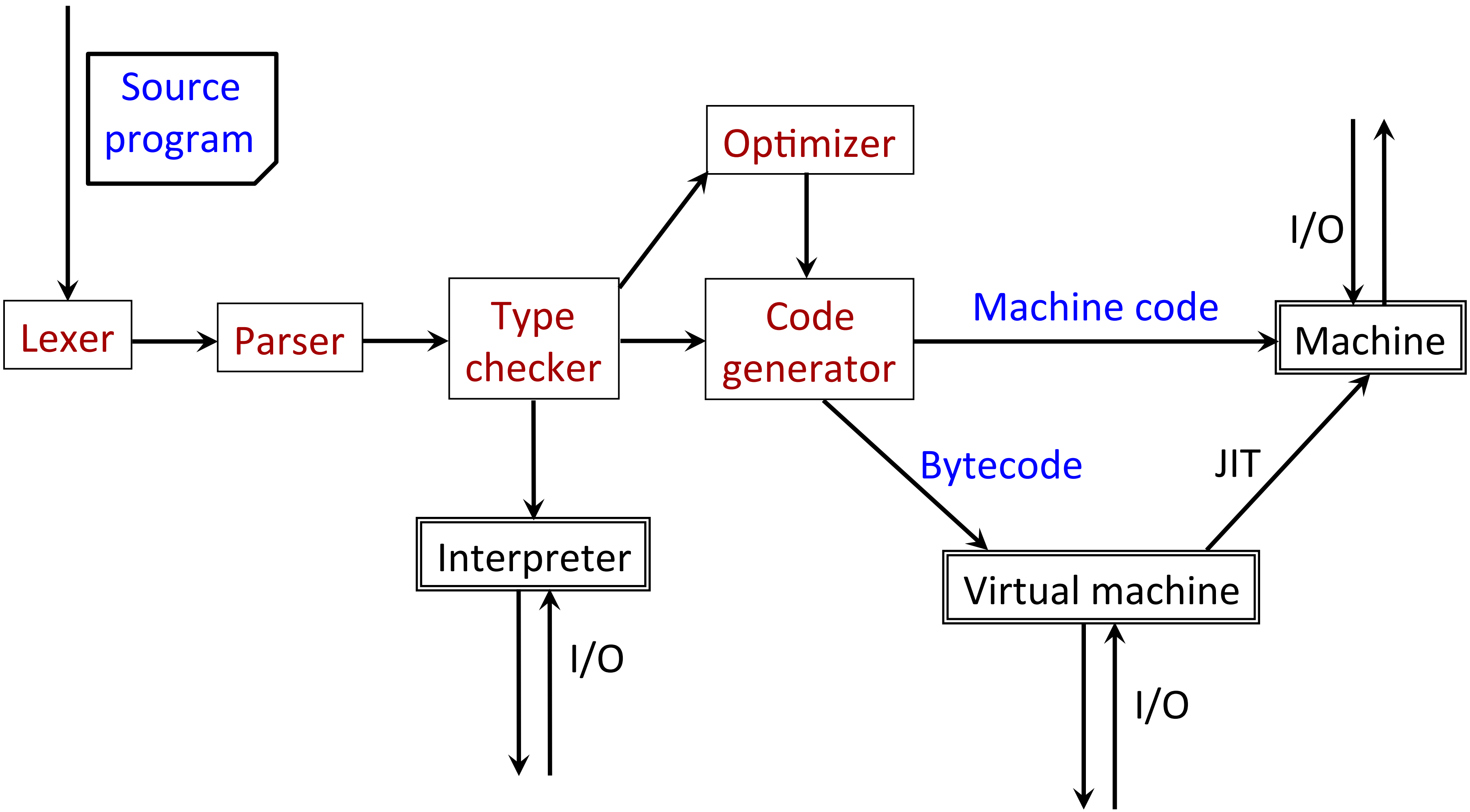
- to assembly (this is actually Java bytecode):

```
iconst_2   // Put integer 2 on stack
istore_1   // Store the top stack value at location 1
iload_1    // Put the value at location 1 on stack
bipush 7   // Put the value 7 on the stack
iadd       // Add two top stack values together
istore_1   // The sum, on top of stack, stored at location 1
```

- to machine language:

```
00101001010110
01001010100101
```

Implementing a Programming Language – How to Undo the Abstraction



Lexical Analysis

From a stream of characters

if (a == b) return;

to a stream of *tokens*

keyword[‘if‘]
symbol[‘(‘]
identifier[‘a‘]
symbol[‘==‘]
identifier[‘b‘]
symbol[‘)‘]
keyword[‘return‘]
symbol[‘;‘]

Syntactic Analysis (Parsing)

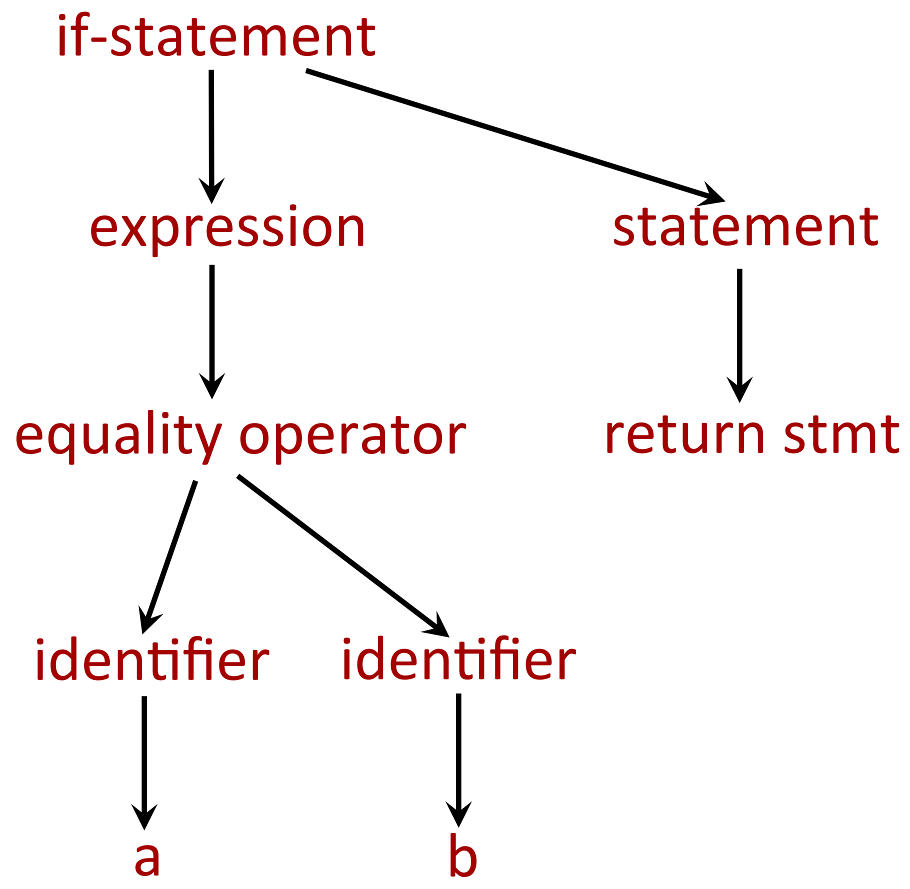
From a stream of characters

if (a == b) return;

to a stream of *tokens*

keyword[`if`]
symbol[`(`]
identifier[`a`]
symbol[`==`]
identifier[`b`]
symbol[`)`]
keyword[`return`]
symbol[`;`]

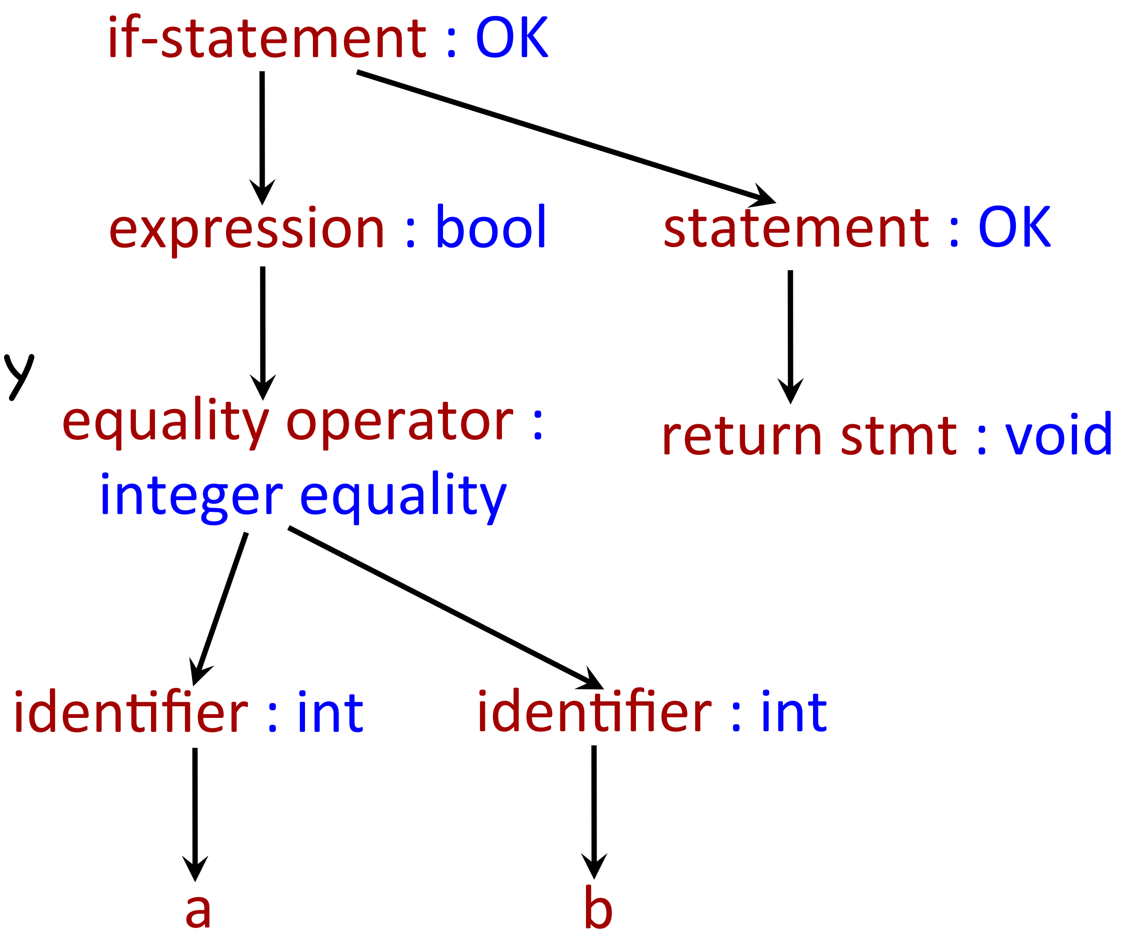
to a syntax tree (parse tree)



Type Checking

if (a == b) return;

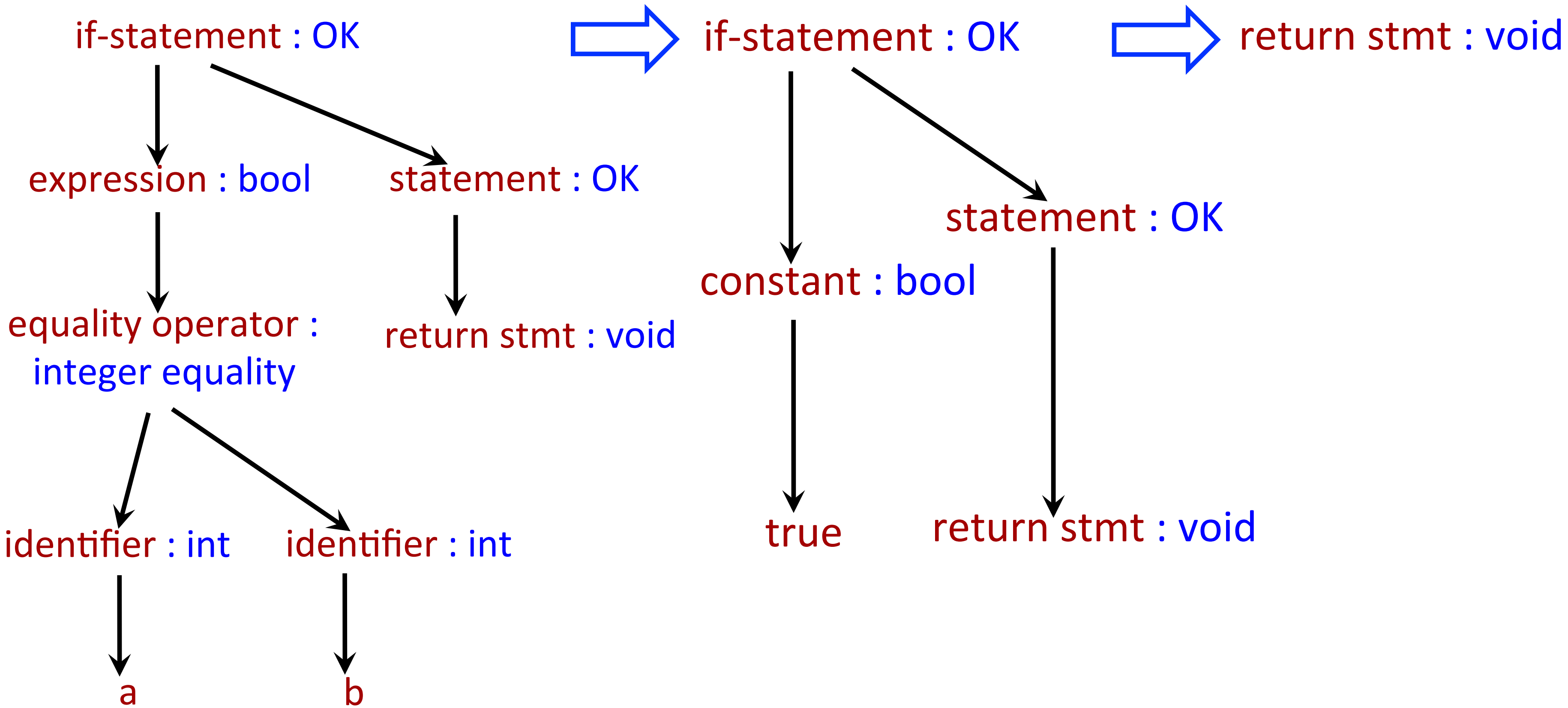
Annotate syntax tree with types, check that types are used correctly



Optimization

```
int a = 10;
int b = 20 - a;
if (a == b) return;
```

Constant propagation can deduce that always a==b, allowing the optimizer to transform the tree:



Code Generation

Code generation is essentially undoing abstractions, until code is executable by some target machine:

- Control structures become jumps and conditional jumps to labels (essentially goto statements)
- Variables become memory locations
- Variable names become addresses to memory locations
- Abstract data types etc. disappear. What is left is data types directly supported by the machine such as integers, bytes, floating point numbers, etc.
- Expressions become loads of memory locations to registers, register operations, and stores back to memory

Phases of Compilation/Execution Characterized by Errors Detected

- Lexical analysis:

  ```
  5abc
  a === b
  ```

- Syntactic analysis:

  ```
  if + then;
  int f(int a];
  ```

- Type checking:

  ```
  void f(); int a; a + f();
  ```

- Execution time:

  ```
  int a[100]; a[101] = 5;
  ```

Compiling and Interpreting (1)

- Typically compiled languages:
  - C, C++, Eiffel, FORTRAN
  - Java, C# (compiled to bytecode)
- Typically interpreted languages:
  - Python, Perl, Prolog, LISP
- Both compiled and interpreted:
  - Haskell, ML, Scheme

Compiling and Interpreting (2)

- Borderline between interpretation and compilation not clear (not that important either)
- Same goes with machine code vs. byte code
- Examples of modern compiling/interpreting/executing scenarios:
  - C and C++ can be compiled to LLVM bytecode
  - Java compiled to bytecode, bytecode interpreted by JVM, unless it is first JITted to native code, which can then be run on a virtual machine such as VMWare