CS 152: *Programming Language Paradigms*

Syntax, Semantics, and Language Design Criteria

Prof. Tom Austin
San José State University

Lab 1 solution
(in class)

Formally defining a language

Two aspects of a language:

- *Syntax* – structure of a program
- *Semantics* – meaning of a program

The two parts of syntax

- *Lexemes* or *tokens* – the "words" of the language
- *Grammar* – the way that words can be ordered

How a compiler works

Tokens are the "words"
of the language.

How a compiler works

```
if (x < 42) {
   y++;
} else {
  y = 42;
}
```

source code → **Lexer/ Tokenizer** → tokens

```
"if" "("  "x" "<"
"42" ")" "{" "y"
"++" ";" "}"
"else" "{" "y"
"=" "42" ";" "}"
```

Tokens are the "words" of the language.

How a compiler works

```
if (x < 42) {
   y++;
} else {
  y = 42;
}
```

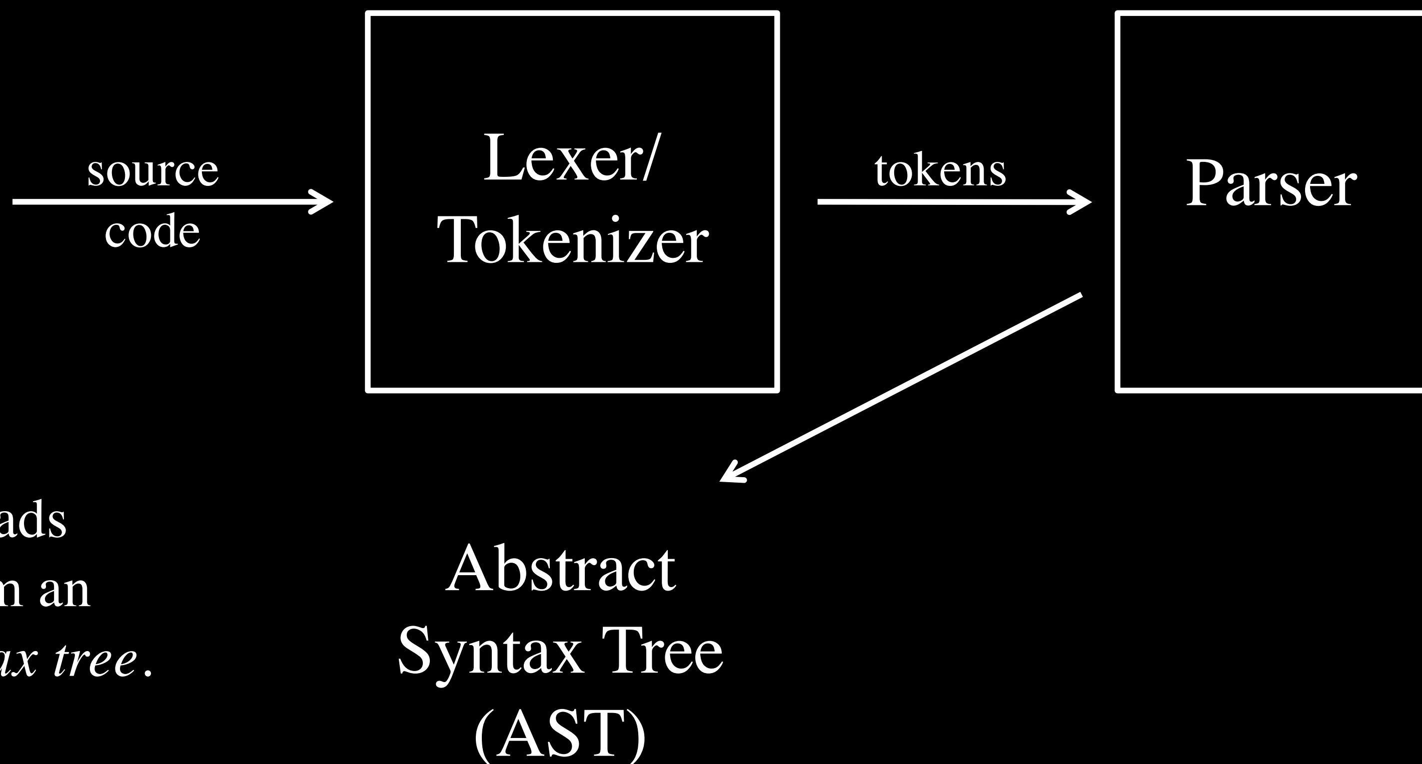
source code → **Lexer/ Tokenizer** → tokens

```
"if" "(" "x" "<"
"42" ")" "{" "y"
"++" ";" "}"
"else" "{" "y"
"=" "42" ";" "}"
```

Types of tokens:

- Identifiers
- Numbers
- Reserved words
- Special characters

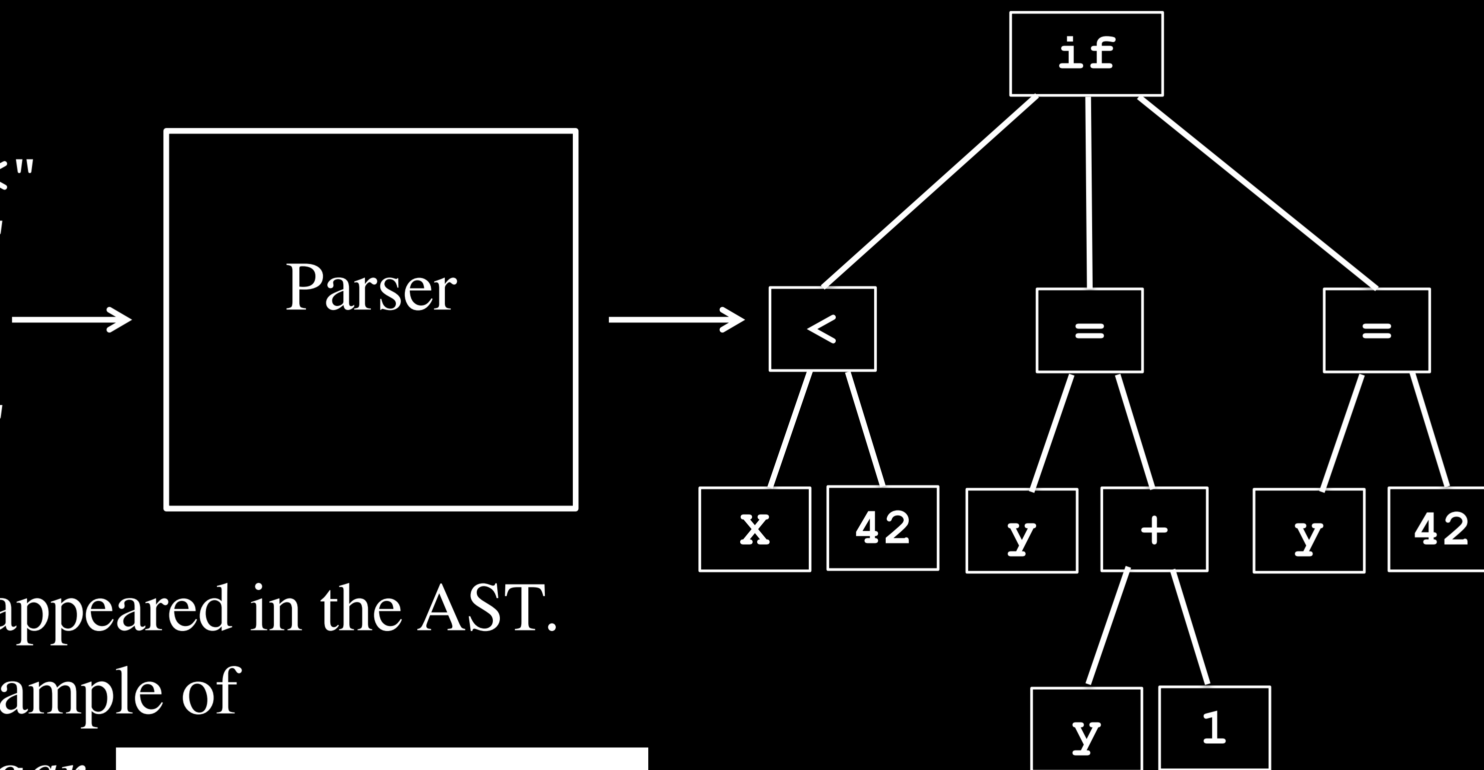
How a compiler works



The parser reads tokens to form an *abstract syntax tree*.

Parsing Example

```
"if" "(" "x" "<"
"42" ")" "{" "y"
"++" ";" "}"
"else" "{" "y"
"=" "42" ";" "}"
```



y++ has disappeared in the AST.
'++' is an example of
syntactic sugar.

Formally defining language syntax

Context-free grammars define the structure of a language.

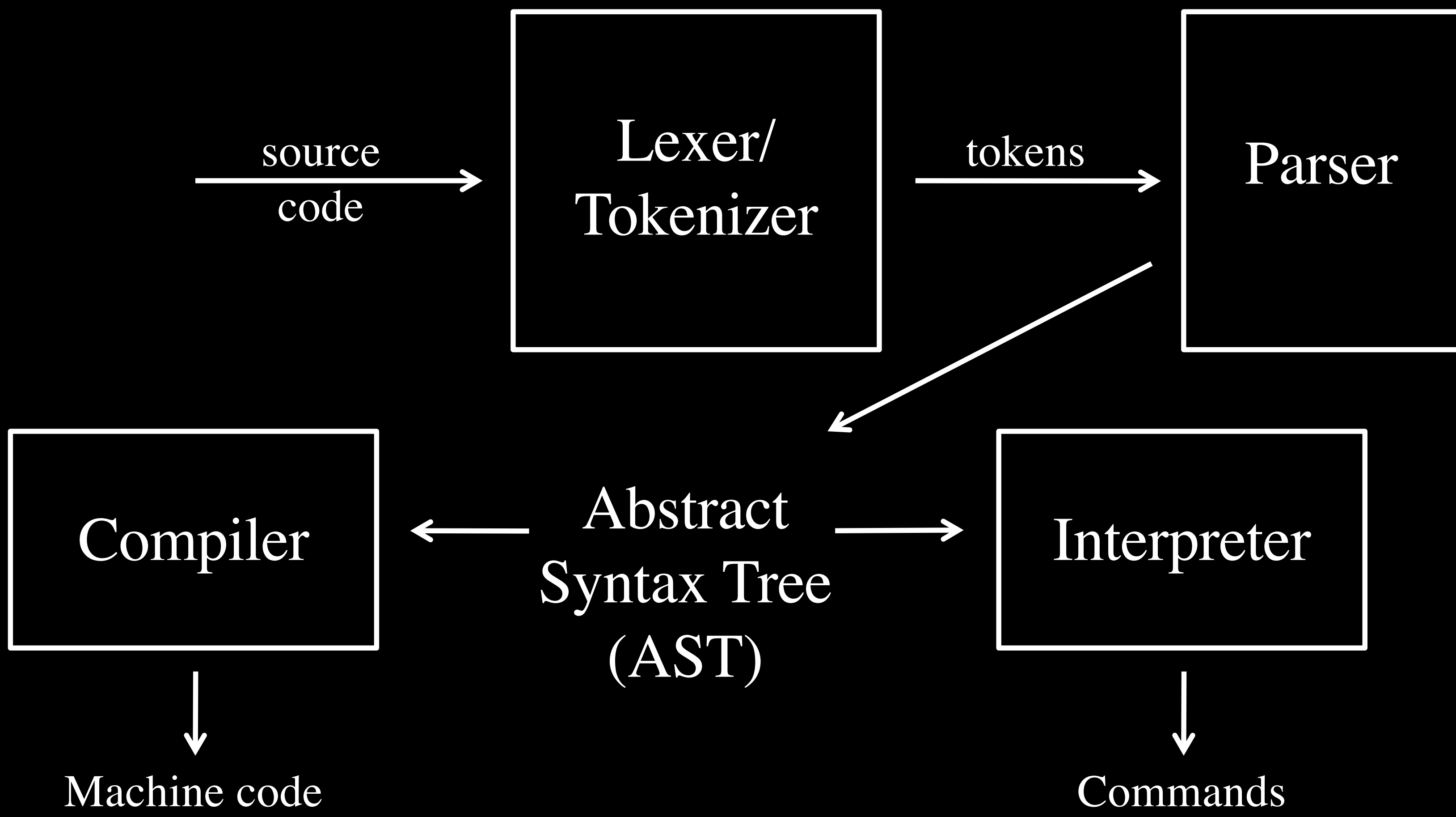
Backas-Naur Form (BNF) is a common notation.

Context-free grammar for math expressions (in BNF notation)

```
<expr> -> <expr> + <term>
        | <expr> - <term>
        | <term>

<term> -> <term> * <factor>
        | <term> / <factor>
        | <factor>
```

How a compiler works



Compilers and interpreters derive *meaning* from ASTs to turn programs into actions.

Covered another day

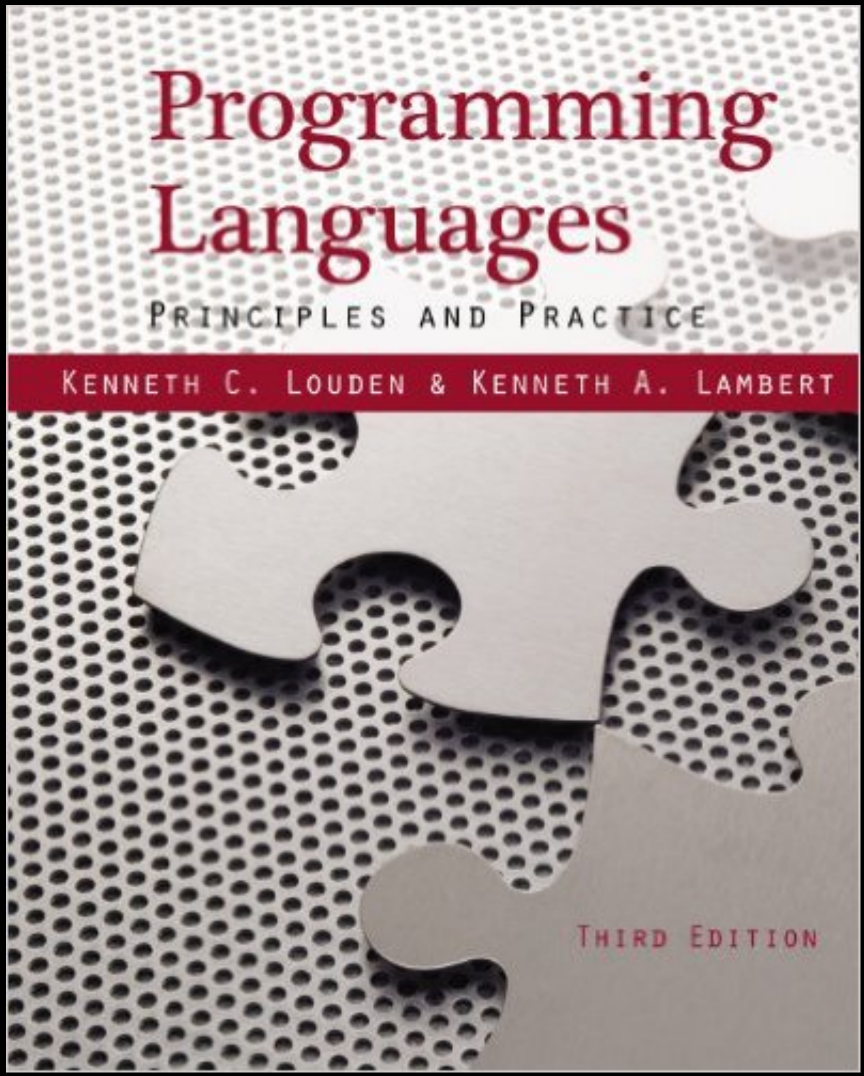
Formally defining language meaning:

- Operational semantics
- Denotational semantics
- Axiomatic semantics

Judging a language

Louden & Lambert's Design Criteria

1. Efficiency
2. Regularity
3. Security
4. Extensibility



Efficiency

- Machine efficiency
  - tips to the compiler
- Programmer efficiency
  - ease of writing programs
  - *expressiveness* (conciseness helps)
- Reliability
  - code maintenance

Efficiency

<u>Java:</u>

```
int i = 10;
String s = "hi";
```

<u>Python:</u>

```
i = 10
s = "hi"
```

- **Machine efficiency:**
  Java offers tips to the compiler
- **Programmer efficiency:**
  Python reduces the amount of typing required

Regularity

- **Generality:**
  - avoid special cases
  - favor general constructs
- **Orthogonal design:**
  - different constructs can be combined with no unexpected restrictions
- **Uniformity**
  - similar things look similar
  - different things look different

Bad uniformity example (PHP): Same things look different

Inconsistent function naming:

- `isset()`
- `is_null()`
- `strip_tags()`
- `stripslashes()`

PHP

TRAINING WHEELS WITHOUT THE BIKE

Bad uniformity example (Pascal): Different things look the same

```
function f : boolean;
  begin
  ...
  f := true;
  end;
```

Return value is true

Security

- Stop programmer errors
  - or handle them gracefully
- Strong typing prevents some run-time errors.
- *Semantically-safe* languages
  - stop executing code violating language definition
  - Contrast array handling by Java and by C/C++

Safety (Java vs. Scheme)

Java:

```
int x = 4;
boolean b = true;
if (b) {
   x++;
} else {
   x = x / "2";
}
```

Scheme:

```
(let ([x 4]
      [b #t])
  (if b
    (+ 1 x)
    (/ x "2")))
```

Extensibility

Allows the programmer to add new language constructs easily.

Macros in Scheme are an example.

Before next class

Read Chapter 6 of *Teach Yourself Scheme*.

Lab 2: More Scheme practice

- Download lab2.rkt from the course website
  - Implement `reverse` function
  - Implement `add-two-lists`
  - Implement `positive-nums-only`
- Using Louden & Lambert's criteria, compare Java & Scheme (or two languages of your choice)