CS 152: *Programming Language Paradigms*

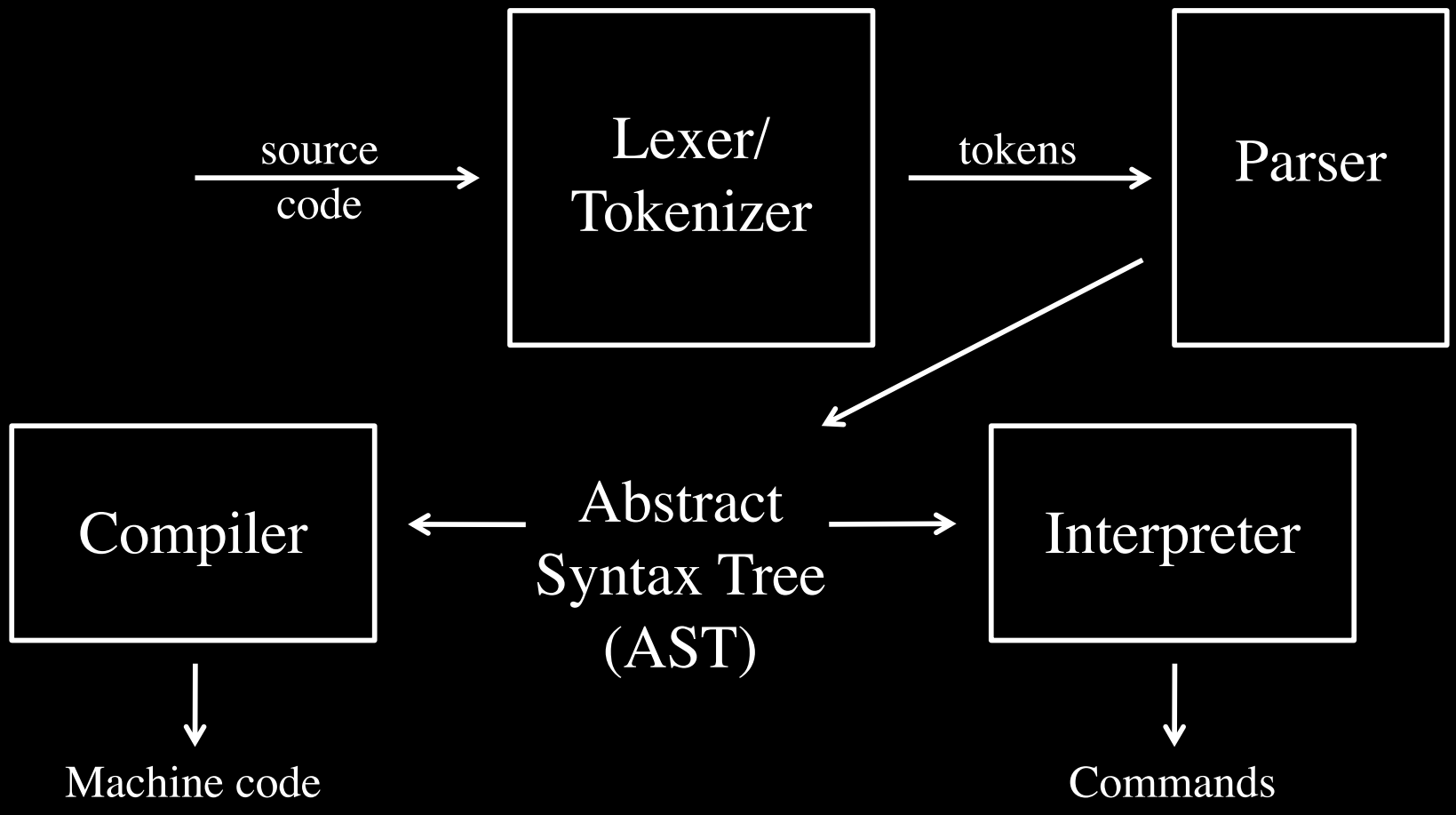
# Syntax & ANTLR

Prof. Tom Austin
San José State University

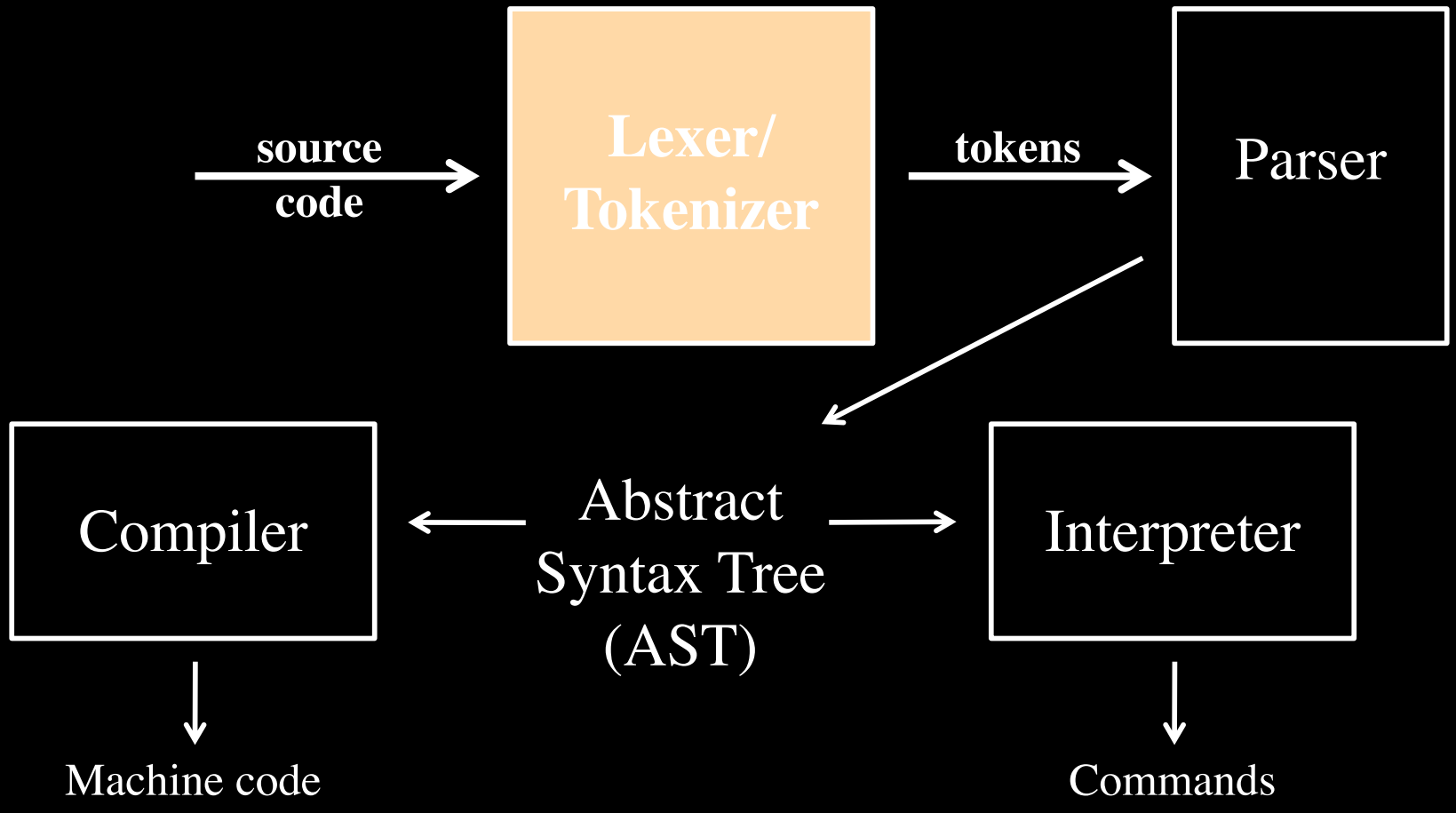
# Syntax vs. Semantics

- Semantics:
  - What does a program mean?
  - Defined by an interpreter or compiler
- Syntax:
  - How is a program structured?
  - Defined by a lexer and parser

# Review: Overview of Compilation



# Tokenization



# Tokenizer

- Converts chars to *words* of the language
- Defined by *regular expressions*
- A variety of lexers exist:
  - Lex/Flex are old and well-established
  - ANTLR & JavaCC work in Java
- Sample lexing rule for integers (in Antlr)

```
INT   : [0-9]+ ;
```

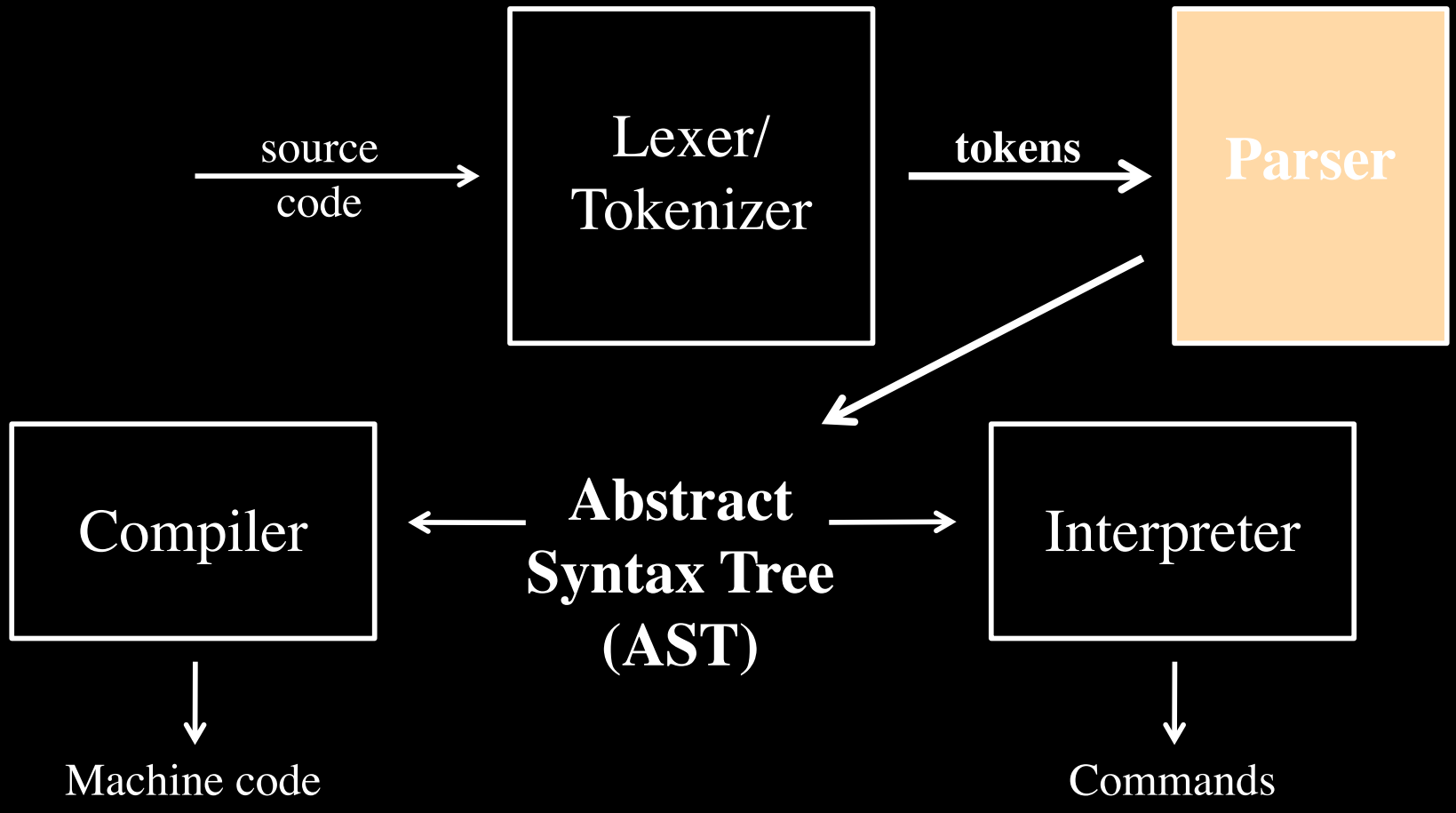
# Categories of Tokens

- Reserved words or keywords
  - e.g. `if`, `while`
- Literals or constants
  - e.g. `123`, `"hello"`
- Special symbols
  - e.g. `";"`, `"<="`, `"+"`
- Identifiers
  - e.g. `balance`, `tyrionLannister`

# Lexing in ANTLR (v. 4)

(in class)

# Parsing


source
code
Lexer/
Tokenizer
tokens
Parser
Compiler
Abstract
Syntax Tree
(AST)
Interpreter
Machine code
Commands

# Parser

- Takes tokens and combines them into *abstract syntax trees* (ASTs)
- Defined by *context free grammars*
- Parsers can be divided into
  - bottom-up/shift-reduce parsers
  - top-down parsers

# Context Free Grammars (CFGs)

- Grammars specify a language
- Backus-Naur form is a common format

```
Expr -> Number
      | Number + Expr
```

- **Terminals** cannot be broken down further.
- **Non-terminals** can be broken down into further phrases.

# Sample grammar

```
expr -> expr + expr
      | expr - expr
      | ( expr )
      | number
number -> number digit
        | digit
digit -> 0 | 1 | 2 | … | 9
```

# Bottom-up Parsers

- Also known as shift-reduce parsers
  - shift tokens onto a stack
  - reduce to a non-terminal
- LR: left-to-right, rightmost derivation
  - Look-Ahead LR parsers (LALR)
    - most common LR parser
    - YACC/Bison are examples

Though generally considered to be more powerful, LALR parsers seem to be fading from popularity.

Top-down (LL) parsers are becoming more widely used.

# Top-down parsers

- Non-terminals are expanded to match incoming tokens.
- LL: left-to-right, leftmost derivation
- LL(k) parsers
  - look ahead k elements to decide on rule to use
  - example: JavaCC
- LL(1) parsers are of special interest:
  - Easy to write/fast execution time
  - Some languages are designed to be LL(1)

# LL(1) parsers

- Easy to write
- fast execution time
- Some languages are designed to be LL(1)

# ANTLR

- ANTLR v. 1-3 were LL(*)
  - Similar to LL(k), but look ahead as far as needed
- ANTLR v. 4 is Adaptive LL(*), or ALL(*)
  - Allows *left-recursive* grammars that were not previously possible with LL parsers. http://www.antlr.org/papers/allstar-techreport.pdf
  - Sample left-recursive grammar:

```
expr -> expr + expr | num
```

# Parsing with ANTLR

(in-class)

# Lab: Getting to know ANTLR

Write a calculator using ANTLR. Details in Canvas, starter code on course website.