CS 252:

*Advanced Programming Language Principles*

# Parsing Combinators

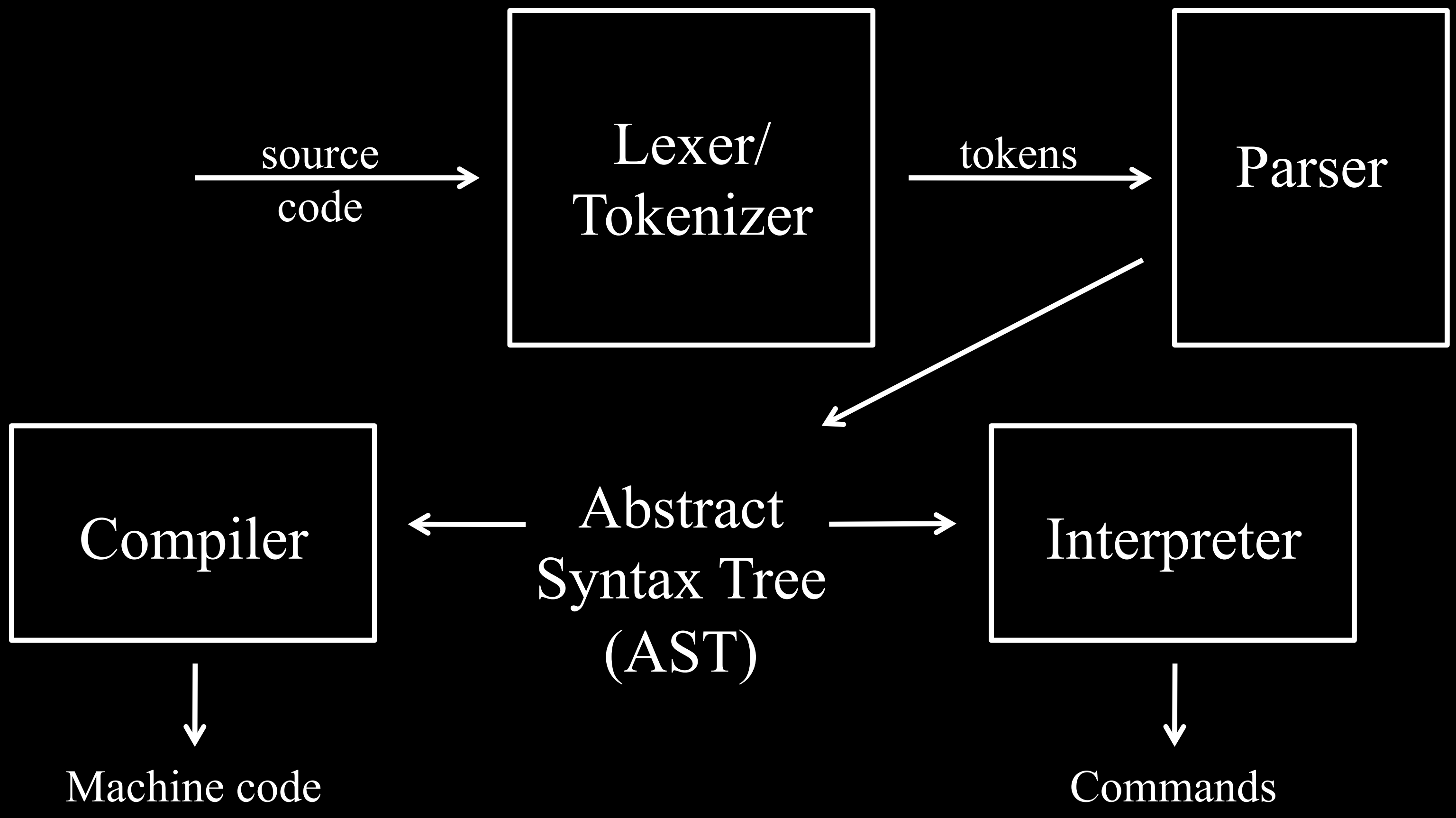
Prof. Tom Austin

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



source code

Lexer/ Tokenizer

tokens

Parser

Abstract Syntax Tree (AST)

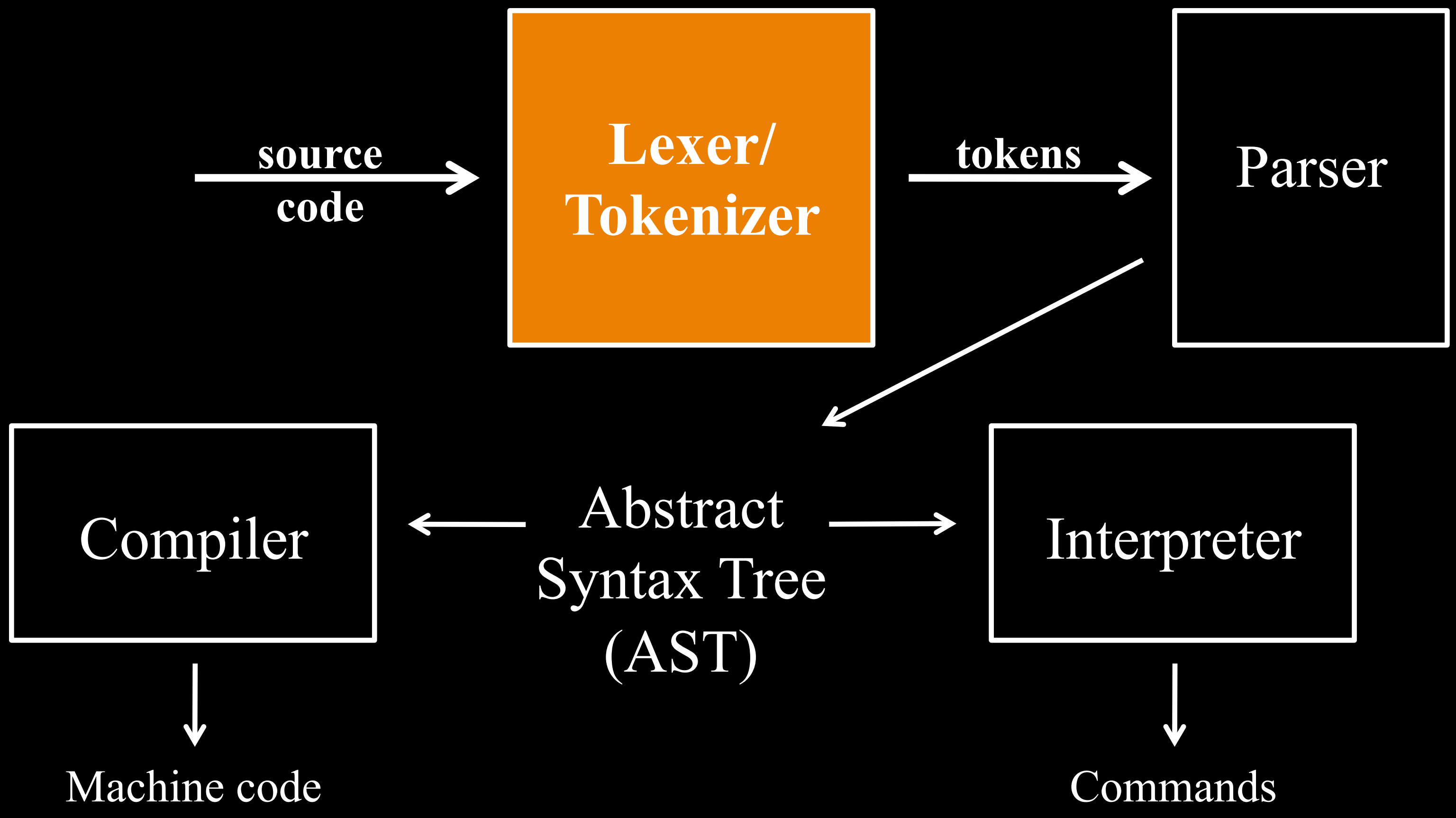
Compiler

Interpreter

Machine code

Commands

# Tokenization



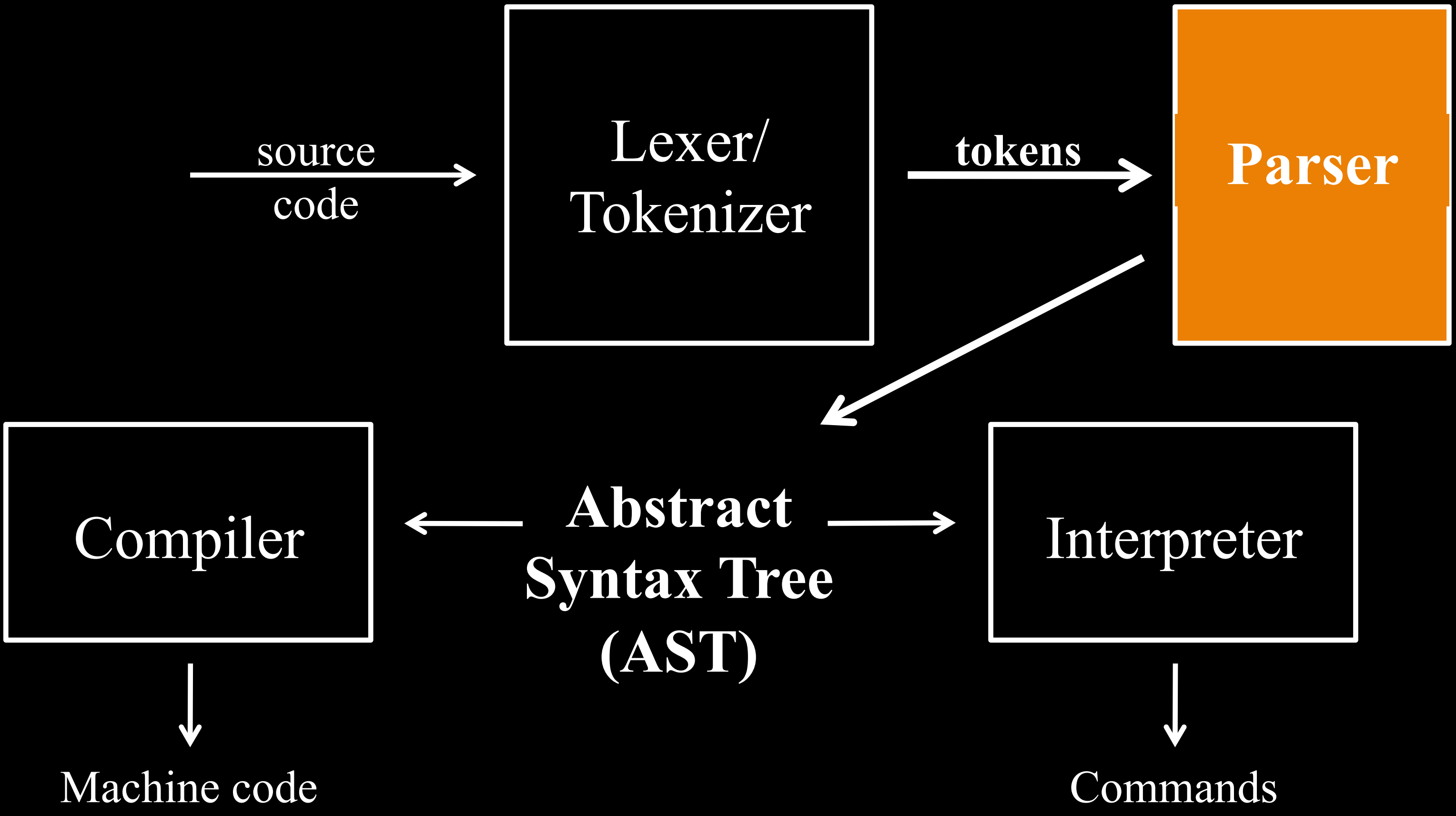
# Tokenization

- Converts characters to the *words* of the language.
- Popular lexers:
  - Lex/Flex (C/C++)
  - ANTLR & JavaCC (Java)
  - ***Parsec*** (Haskell)

# 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`

# Parsing



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

# Parsing

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

# Context Free Grammars

- Grammars specify a language
- Backus-Naur form 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

- a.k.a. **shift-reduce parsers**
  1. shift tokens onto a stack
  2. reduce to a non-terminal.
- **LR**: left-to-right, rightmost derivation
- Look-Ahead LR parsers (**LALR**)
  - most popular style of LR parsers
  - YACC/Bison
- Fading from popularity.

# Top-down parsers

- Non-terminals expanded to match tokens.
- **LL**: left-to-right, leftmost derivation
- **LL(k)** parsers look ahead k elements
  - example LL(k) parser: JavaCC
  - LL(1) parsers are of special interest

# Parser combinators

- Combine simpler parsers to make a more complex parser
- Example in Parsec:

```
num :: GenParser Char st String
num = many1 digit
```

Type of result

```haskell
import Text.ParserCombinators.Parsec

num :: GenParser Char st String
num = many1 digit

main = do
  print $ parse num "example 1" "42"
```

```
import Text.ParserCombinators.Parsec

num :: GenParser Char st Integer
num = do
  str <- many1 digit
  return $ read str

main = do
  print $ parse num "example 2" "42"
```

# Some useful functions

- `many/many1`: 0/1 or more of …
- `noneOf`: Anything but …
- `spaces`: whitespace characters
- `char`: the character ...
- `string`: the string …

# CSV parser (1st attempt)

(in-class)

```
Year,Make,Model,Length
1997,Ford,E350,2.34
2000,Mercury,Cougar,2.38
```

# Example Using <|>, <?>, and try

```
eol = try (string "\n\r")
       <|> string "\n"
       <?> "end of line"
```

If you can't match, rewind.

# CSV parser (2nd attempt)

(in-class)

```
Year,Make,Model,Length
1997,Ford,E350,2.34
2000,Mercury,Cougar,2.38
```

# JSON example

```
{ name: "Complex number example",
  nums: [
    { real: 42, imaginary: 1 },
    { real: 30, imaginary: 0 },
    { real: 15, imaginary: 7 } ],
  knownIssues: null,
  verified: false }
```

# Lab: Parsec

This lab is available in Canvas.

Starter code is available on the course website.