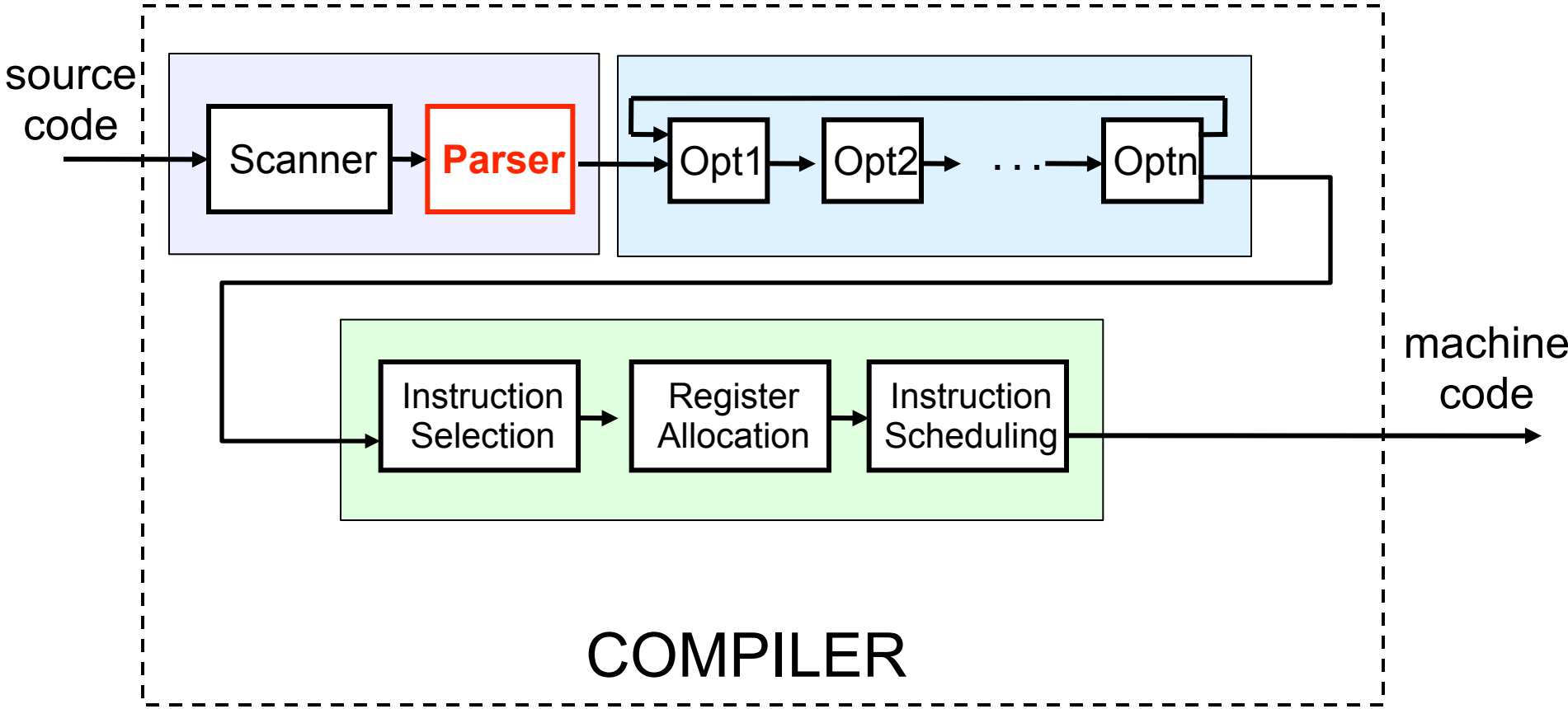
Syntactic Analysis ("parsing")

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

The Big Picture Again



Syntactic Analysis

- Lexical Analysis was about ensuring that we extract a set of valid words (i.e., tokens/lexemes) from the source code
- But nothing says that the words make a coherent sentence (i.e., a program that can be compiled)
- Example:
  - "if while i == == == 12 + endif abcd"
  - Lexer will produce a stream of tokens: **<TOKEN_IF> <TOKEN_WHILE> <TOKEN_NAME, "i"> <TOKEN_EQUAL> <TOKEN_EQUAL> <TOKEN_EQUAL> <TOKEN_INTEGER," 12"> <TOKEN_PLUS, "+"> <TOKEN_ENDIF> <TOKEN_NAME, "abcd">**
  - This program is **lexically correct**, but **syntactically incorrect**
    - Just like in English "apple me ate tree tree" is lexically correct but syntactically incorrect

Grammar

- Question: How do we determine that a sentence is syntactically correct?
- Answer: We check against a grammar!
- A grammar consists of rules that determine which sentences are correct
- Example in English:
  - A sentence must have a verb
- Example in C:
  - A “{“ must have a matching “}”

Grammar

- Regular expressions are one way we have seen for specifying a set of rules/patterns
- Unfortunately they are not powerful enough for describing the syntax of programming languages
- Example:
  - If we have 10 ‘{‘ then we must have 10 ‘}’
  - We can’t implement this with regular expressions because they do not have memory!
    - No way of counting and remembering counts
- Therefore we need a more powerful tool
- This tool is called Context-Free Grammars (CFG)
  - And some additional mechanisms

Context-Free Grammars

- A context-free grammar (CFG) consists of a set of production rules
- Each rule describes how a non-terminal symbol can be “replaced”/“expanded”/“rewritten” by a string that consists of non-terminal symbols and terminal symbols
  - Terminal symbols are lexical tokens (i.e., valid “words”)
  - Rules are written with regex-like syntax
- Rules can then be applied recursively
- Eventually one reaches a string of only terminal symbols (unless a syntax error is found)
- This string is then proven syntactically correct according to the grammatical rules!

CFG Example

- Set of non-terminals: A, B, C (uppercase initial)
- Start non-terminal: S (uppercase initial)
- Set of terminal symbols: a, b, c, d (lowercase initial)
- Empty symbol: ε
- Set of production rules:

  S → A | BC
  A → Aa | a
  B → bBCb | b
  C → dCcd | c

- We can now start producing syntactically valid strings by doing derivations
- Examples (rewriting a non-terminal each time):

S → BC → bBCbC → bbCbC → bbdCcdbC → bbdccdbC → bbdccdbc

S → A → Aa → Aaa → Aaaa → aaaa

A Grammar for Expressions

Expr	➔ Expr Op Expr
Expr	➔ Number | Identifier
Identifier	➔ Letter | Letter Identifier
Letter	➔ ‘a’-‘z’
Op	➔ ‘+’ | ‘-‘ | ‘*’ | ‘/‘
Number	➔ Digit Number | Digit
Digit	➔ ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

Expr ➔ Expr Op Expr ➔ Number Op Expr ➔

Digit Number Op Expr ➔ 3 Number Op Expr ➔ 34 Op Expr ➔

34 * Expr ➔ 34 * Identifier ➔ 34 * Letter Identifier ➔

34 * a Identifier ➔ 34 * a Letter ➔ 34 * ax

Ambiguous Grammar?

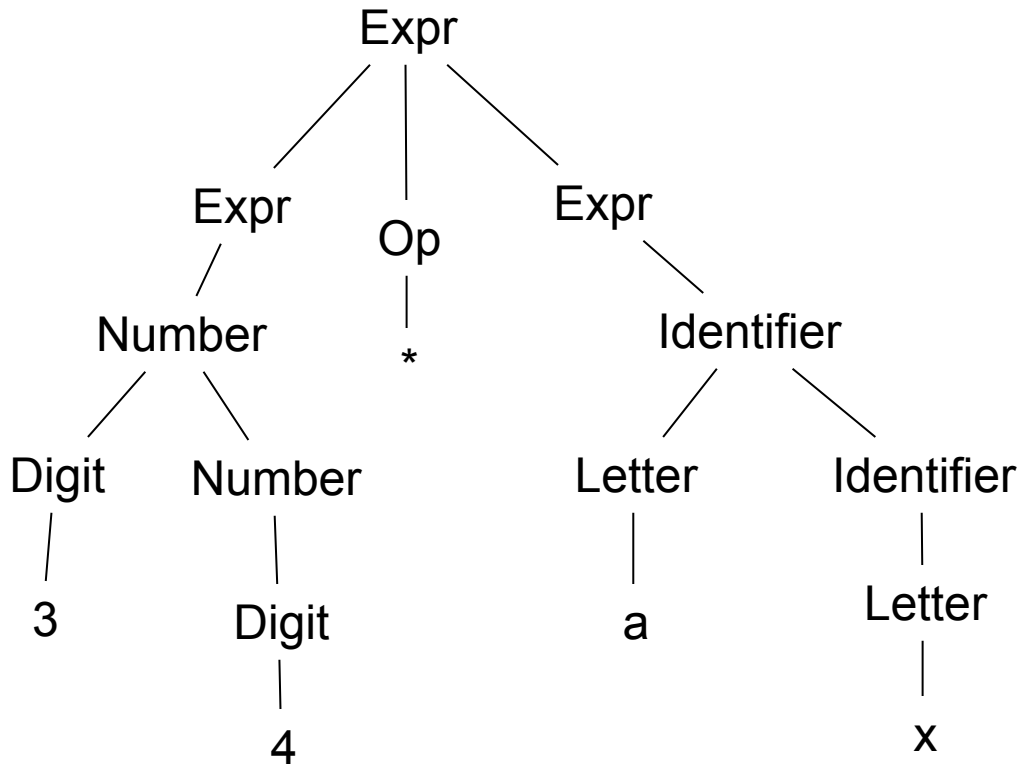
- The grammar on the previous slide is not what you'd write for a real/production compiler
- That grammar is **ambiguous**
  - It can lead to different parse trees for the same string, which causes all kinds of problems
- There are known techniques to make grammars non-ambiguous
  - They make grammars larger and more difficult for humans to read
- In these course, we ignore ambiguity

What is Parsing?

- What we just saw is the process of, starting with the start symbol and, through a sequence of rule derivations, obtain a string of terminal symbols: derivation
  - We could generate all correct programs (it's an infinite set though)
- **Parsing**: the other way around
  - Given a string of non-terminals, discover a sequence of rule derivations that produce this particular string
- When we say we can't parse a string, we mean that we can't find any legal way in which the string can be obtained from the start symbol through derivations
  - We call this a syntax error
- What we want to build is a parser: a program that takes in a string of tokens (terminal symbols) and discovers a derivation sequence or says "syntax error"

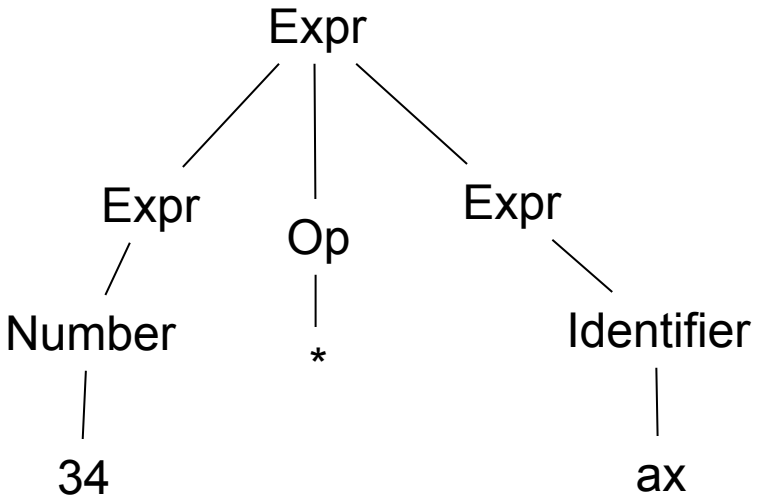
Derivations as Trees

- A convenient and natural way to represent a sequence of derivations is a syntactic tree or parse tree
- Example: Expr ➔ Expr Op Expr ➔ Number Op Expr ➔ Digit Number Op Expr ➔ 3 Number Op Expr ➔ 34 Op Expr ➔ 34 * Expr ➔ 34 * Identifier ➔ 34 * Letter Identifier ➔ 34 * a Identifier ➔ 34 * a Letter ➔ 34 * ax



Derivations as Trees (2)

- Often, we draw trees without the full derivations (i.e., we aggregate trivial subtrees)
- Example:



Example #1

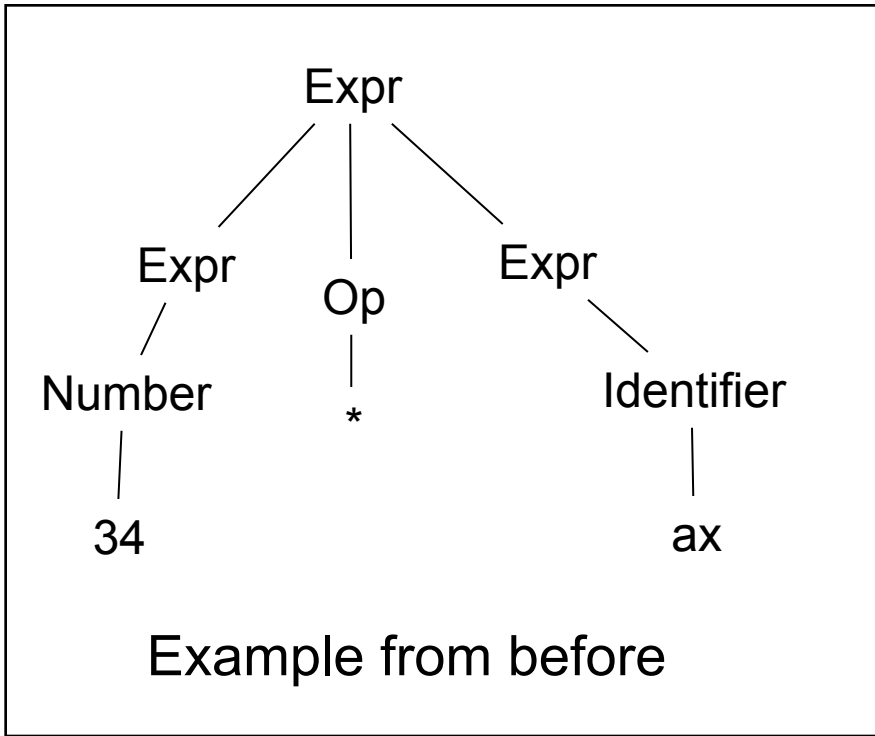
- Consider the CFG:

  S → ‘(‘ L ‘)’ | ‘a’

  L → L ‘,’ S | S

Draw parse trees for:

(a, a)



Example from before

Example #1 (Solution)

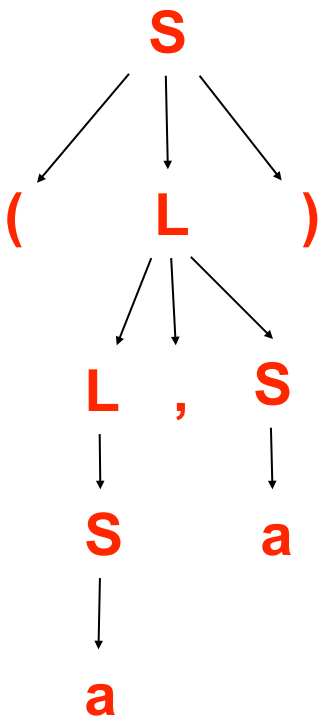
- Consider the CFG:

  S → ‘(‘ L ‘)’ | ‘a’

  L → L ‘,’ S | S

Draw parse trees for:

(a, a)



In-Class Exercise

- Write a CFG for the language of all possible non-empty strings over the alphabet {‘0’, ‘1’}
  - e.g., 0, 1, 01, 10, 00000, 1010101, etc.
  - Yes, we can do this with a regular expression, but let’s do a CFG anyway

In-Class Exercise (solution)

- Write a CFG for the language of all possible, non-empty strings over the alphabet {‘0’, ‘1’}

S -> 0 S | 1 S | 0 | 1

The above is recursive (a string in the language is either a 0 or a 1, or starts with a 0 or a 1 and is then followed by a string in the language)

Example CFG for a for loop

ForStatement → ‘for’ ‘(‘ StmtCommaList ‘;’
ExprCommaList ‘;’ StmtCommaList ‘)’
‘{‘ StmtSemicList ‘}’

StmtCommaList → ε | Stmt | Stmt ‘,’ StmtCommaList

ExprCommaList → ε | Expr | Expr ‘,’ ExprCommaList

StmtSemicList → ε | Stmt | Stmt ‘;’ StmtSemicList

Expr → . . .

Stmt → . . .

Full Language Grammar Sketch

Program ➔ VarDeclList FuncDeclList

VarDeclList ➔ ε | VarDecl | VarDecl VarDeclList

VarDecl ➔ Type IdentCommaList ‘;’

IdentCommaList ➔ Ident | Ident ‘,’ IdentCommaList

Type ➔ int | char | float

FuncDeclList ➔ ε | FuncDecl | FuncDecl FuncDeclList

FuncDecl ➔ Type Ident ‘(‘ ArgList ‘)’ ‘{‘ VarDeclList StmtList ‘}’

StmtList ➔ ε | Stmt | Stmt StmtList

Stmt ➔ Ident ‘=‘ Expr ‘;’ | ForStatement | ...

Expr ➔ ...

Ident ➔ ...

Using * notations

Program ➔ VarDeclList FuncDeclList

VarDeclList ➔ VarDecl*

VarDecl ➔ Type IdentCommaList ';'

IdentCommaList ➔ Ident (',' Ident)*

Type ➔ int | char | float

FuncDeclList ➔ FuncDecl*

FuncDecl ➔ Type Ident '(' ArgList ')' '{' VarDeclList StmtList '}'

StmtList ➔ Stmt*

Stmt ➔ Ident '=' Expr ';' | ForStatement | ...

Expr ➔ ...

Ident ➔ ...

Real-world CFGs

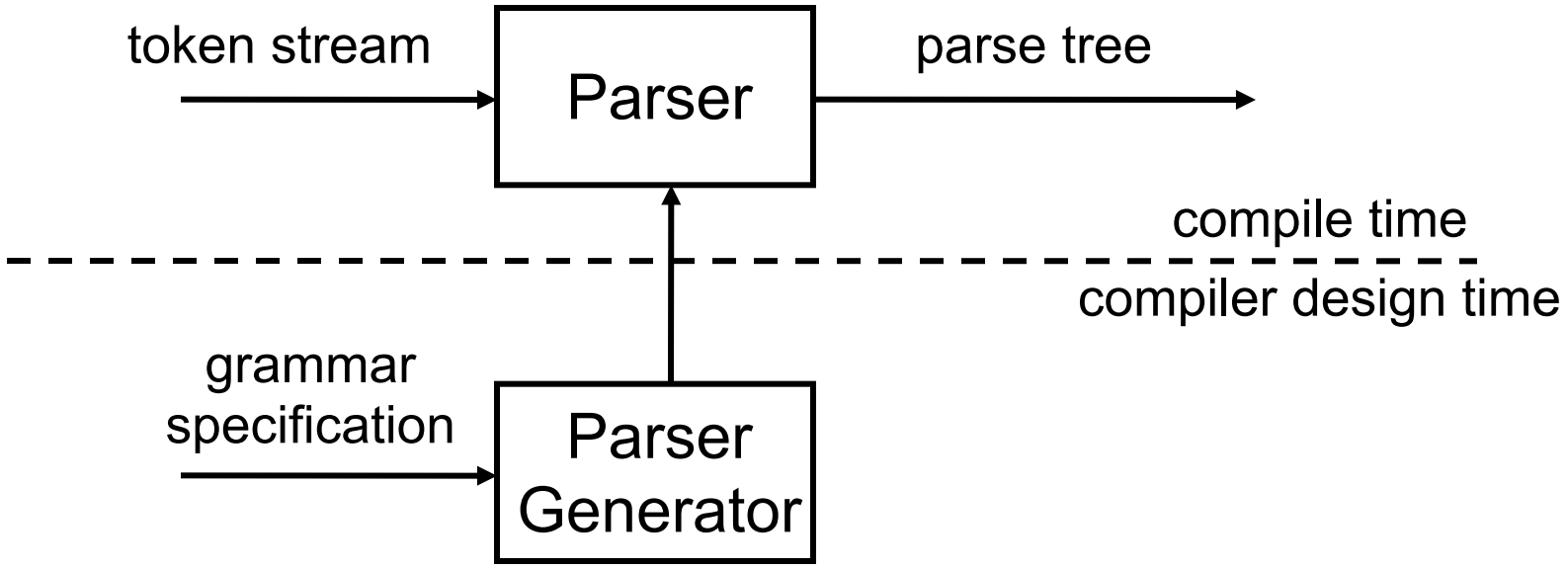
- Some sample grammars found on the Web
  - LISP: 7 rules
  - PROLOG: 19 rules
  - Java: 30 rules
  - C: 60 rules
  - Ada: 280 rules
- LISP is particularly easy to parse because
  - No operators, just function calls
  - Therefore no precedence, associativity
- In the Java specification the description of operator precedence and associativity takes 25 pages!
- Note that the numbers above are to be taken with a grain of salt (they depend on how the grammar is written)

How do we build a Parser?

- This could take one month of a graduate course, as there are many approaches, many algorithms, many challenges
- The (amazing) bottom-line: Given a grammar, provided this grammar abides by a few constraints, we know how to generate the code for a parser that, for any input string, will either say “syntax error” or produce a parse tree

- There is no way we can get into this deeply in this course, so we’re just going to accept that ANTLR does it all for us!

Parser Generator



So What Now?

- Lexing
  - We come up with a definition of the tokens as regular expressions
  - We build a lexer using a tool
- Parsing
  - We come up with a definition of the syntax embodied in a CFG
  - We build a parser using a tool
- And just like that, we have the compiler's front-end!

Important Takeaways

- There are patterns in programming languages that cannot be described using only regular expressions
- We need to resort to Context-Free Grammars (CFGs) to describe these patterns
  - Which make it possible to implement complex recursive structures
- Parsing is: checking that a program matches the pattern of the grammar
- You should be able to write a correct CFG

Conclusion

- At this point, we have a “compiler” that will detect lexical and syntactic errors
  - Lexer or parser errors
- If no errors, then it will generate a parse tree
  - Which we can look at on some GUI
- The next step is to actually have the compiler generate code!
  - And detect “semantic” errors (like assigning a string to an int, which is lexically/syntactically correct, but semantically invalid)
- Let’s do some of the posted Practice Problems…
- We will have a Quiz on this module next week