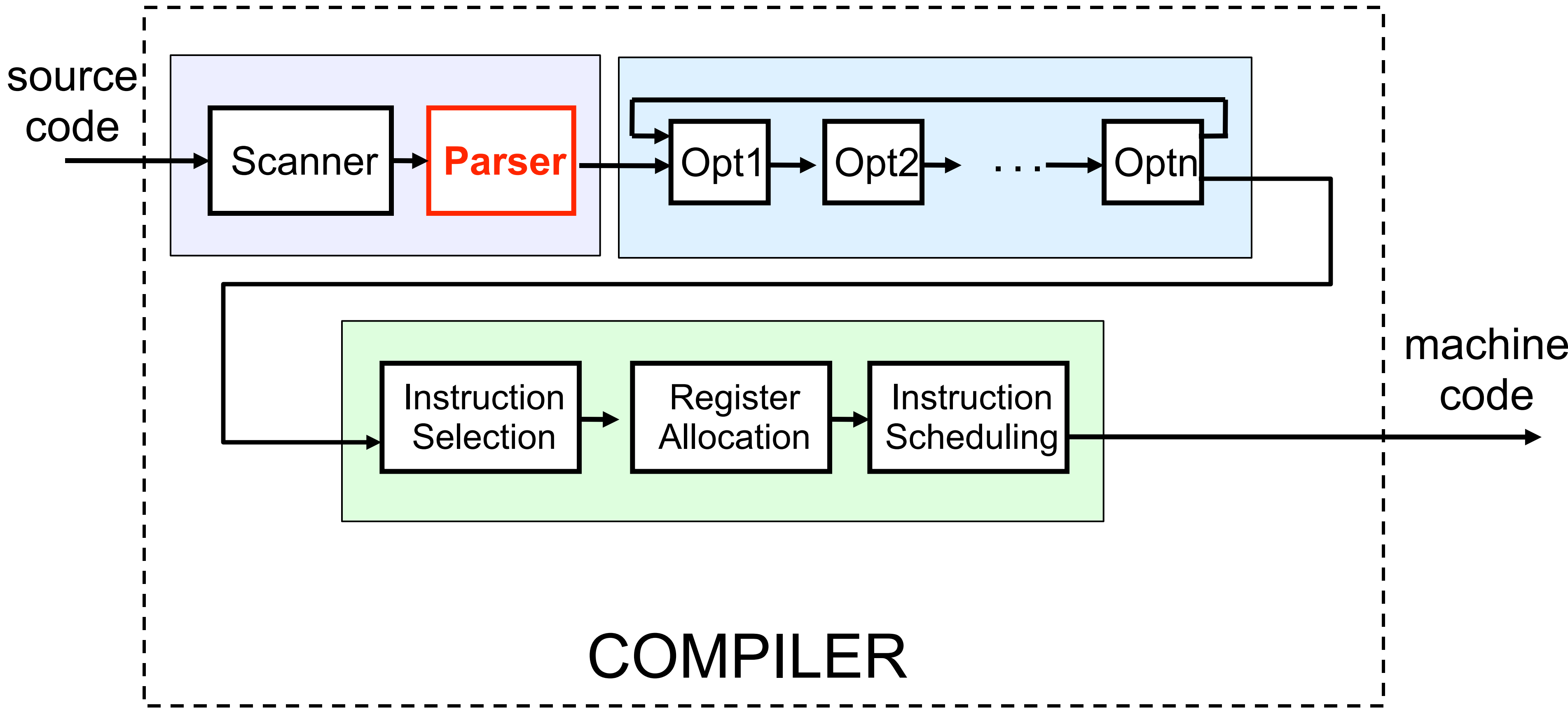
Syntactic Analysis (“parsing”)

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
- Unfortunately they are not powerful enough for describing the syntax of programming languages
- Example:
  - If we have 10 ‘{‘ then me 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
  - 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 really lexical tokens (i.e., valid "words")
  - Rules are written with syntax like regular expressions
- 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

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:

$$S \rightarrow A \mid BC$$
$$A \rightarrow Aa \mid a$$
$$B \rightarrow bBCb \mid b$$
$$C \rightarrow dCcd \mid c$$

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

$S \rightarrow BC \rightarrow bBCbC \rightarrow bbCbC \rightarrow bbdCcdbC \rightarrow bbdccdbC \rightarrow bbdccdbc$

$S \rightarrow A \rightarrow Aa \rightarrow Aaa \rightarrow Aaaa \rightarrow aaaa$

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’

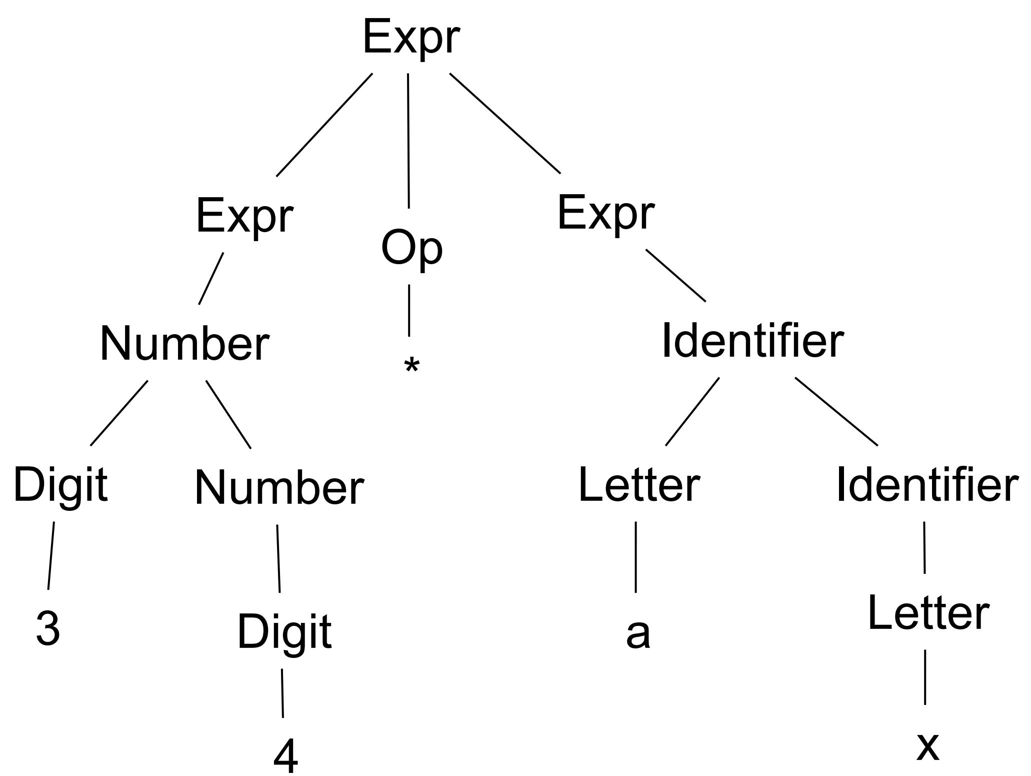
Expr → Expr Op Expr → Number Op Expr →
Digit Number Op Expr → 3 Number Op Expr → 34 Op Expr →
34 * Expr → 34 * Identifier → 34 * Letter Identifier →
34 * a Identifier → 34 * a Letter → 34 * ax

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
  - Give 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"

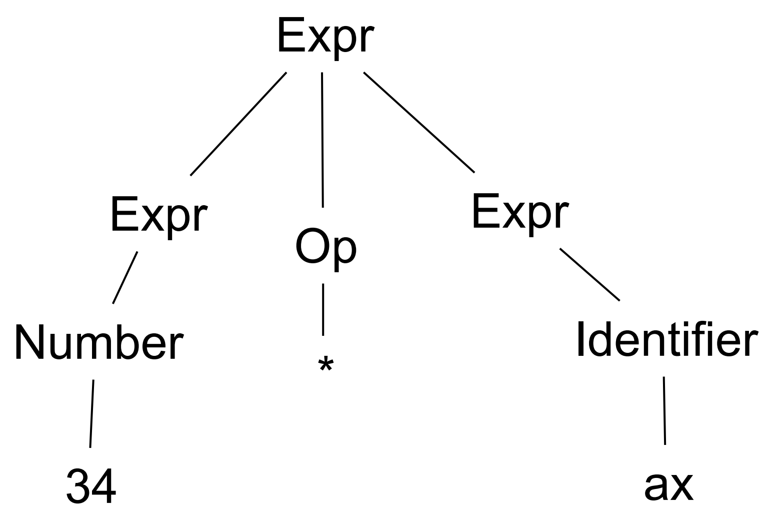
Derivations as Trees

- 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 subtrees)
- Example:



In-class Exercise #1

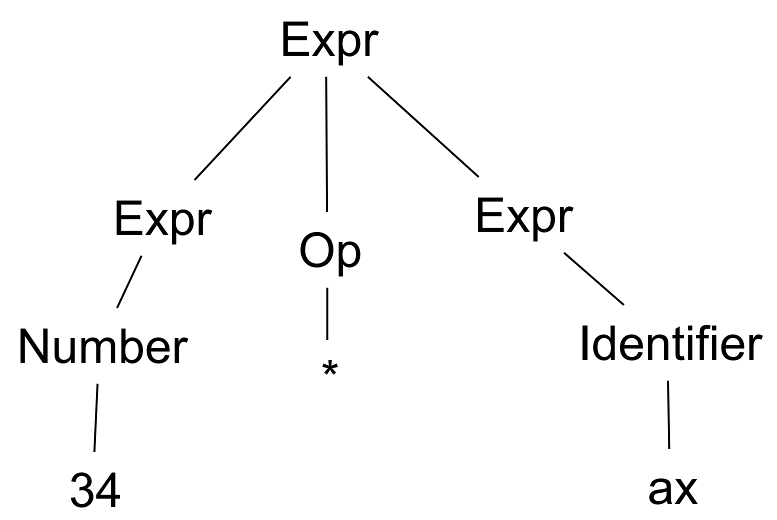
- Consider the CFG:

  S → ‘(‘ L ‘)’ | ‘a’

  L → L ‘,’ S | S

Draw parse trees for:

(a, a)



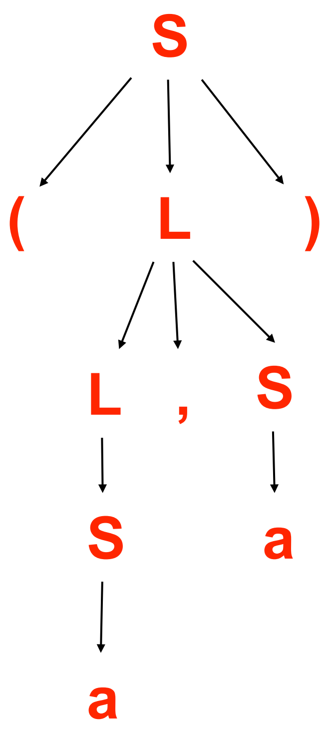
Example from before

In-class Exercise #1 (solution)

- Consider the CFG:

  S → ‘(‘ L ‘)’ | ‘a’

  L → L ‘,’ S | S



Draw parse trees for:

(a, a)

In-class Exercise #2

S $\rightarrow$ ‘(‘ L ‘)’ | ‘a’

L $\rightarrow$ L ‘,’ S | S

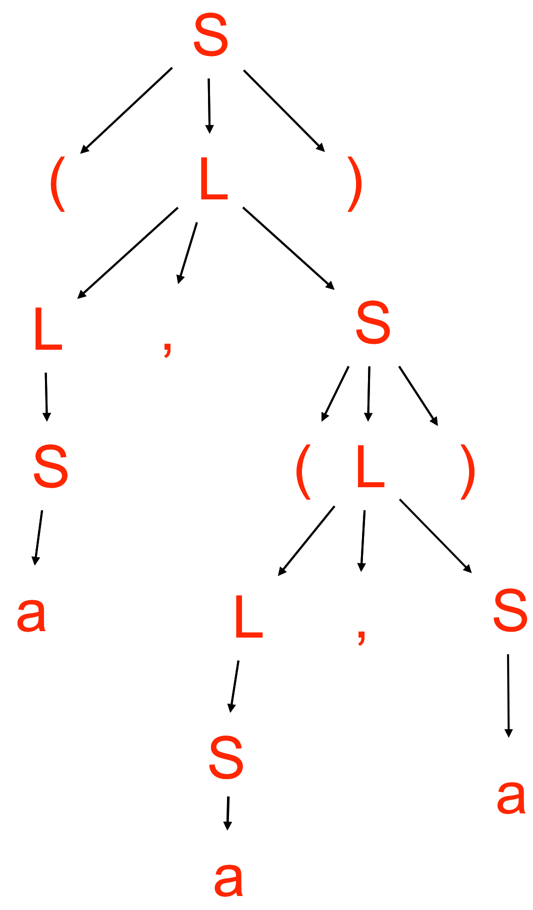
Draw parse tree for: (a, (a, a)) ?

In-class Exercise #2 (solution)

S → '(' L ')' | 'a'

L → L ',' S | S

Draw parse tree for: (a, (a, a)) ?



In-class Exercise #3

- Write a CFG for the language of all possible non-empty strings of 0's and 1's
  - Yes, we can do this with a regular expression, but let's do a CFG anyway

In-class Exercise #3 (solution)

- Write a CFG for the language of all possible, non-empty strings of 0's and 1's

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)

In-class Exercise #4

- Write a CFG for the language of strings that start and end with the same number of 0's
  - No, we could not do this with a regular expression

In-class Exercise #4 (solution)

- Write a CFG for the language of strings that start and end with the same number of 0's

S -> 0 S 0 | L

L -> 1 W 1

W -> 0 W | 1 W | e

Note the recursive "book-ends" pattern of the first rule. Let's looks a wrong (yet commonly written by students) answer…

In-class Exercise #4 (solution)

- Write a CFG for the language of strings that start and end with the same number of 0's

POPULAR BUT WRONG SOLUTION:

S -> L 1 W 1 L

L -> 0 L | ε

The first rule above does not "control" that the numbers of 0's on the left and on the right are the same!!!

In-class Exercise #5

- Write a CFG for the language of well-formed parenthesized expressions
  - (), (()), ()(), (()()), etc.: OK
  - ()),)(, ((()), (((, etc.: not OK

- This is sort of a “do you really understand recursion?” test :)

In-class Exercise #5 (solution)

- Write a CFG for the language of well-formed parenthesized expressions
  - (), (()), ()(), (()()), etc.: OK
  - ()),)(, ((()), (((, etc.: not OK

P → ‘(’‘)’ | PP | ‘(’P‘)’

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!

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 every input string, will either say “syntax error” or build a parse tree

- There is no way we can getting into this deeply in this course, so I’m just going to give you the gist to if
  - ANTLR will do all this for us!

How do we build a Parser?

- Let’s try to see a very high-level view of parsing
  - If you go to grad school you’ll be able to take an in-depth compiler course with all the details
- There are two approaches for parsing:
  - Top-Down: Start with the start symbol and try to expand it using derivation rules until you get the input source code
  - Bottom-Up: Start with the input source code, consume symbols, and infer which rules could be used
- Note: this does not work for all CFGs
  - CFGs must have some properties to be parsable with our beloved parsing algorithms
  - There are tons of results about which algorithm works with which grammars

Top-Down Parsing

- A simple recursive algorithm that searches for the derivations
  - Start with the start symbol
  - Pick one of the rules to expand it an expand it
  - If the leftmost symbol is a non-terminal and matches the current token of the input source, great
  - If there is no match, then backtrack and try another rule
  - Repeat for all non-terminal symbols
  - Success if we get all terminals
  - Failure if we've tried all productions without getting all terminals
- Let's see this on an example

Top-Down Parsing Example

A simple grammar:

(R1) Expr ➔ Number + Expr

(R2) Expr ➔ Number

(R3) Expr ➔ Number * Expr

(R4) Number ➔ 0-9

Top-Down Parsing Example

- Let’s parse string “3*4*5”

A simple grammar:

(R1) Expr ➔ Number + Expr

(R2) Expr ➔ Number

(R3) Expr ➔ Number * Expr

(R4) Number ➔ 0-9

Top-Down Parsing Example

- Let's parse string "3*4*5"
  - Apply **R1**:
    - Apply **R4**: gets the number ("3")
    - Expects a "+", but gets a "*": backtrack

A simple grammar:

(R1) Expr ➔ Number + Expr

(R2) Expr ➔ Number

(R3) Expr ➔ Number * Expr

(R4) Number ➔ 0-9

Top-Down Parsing Example

- Let's parse string "3*4*5"
  - Apply **R1**:
    - Apply **R4**: gets the number ("3")
    - Expects a "+", but gets a "*": backtrack
  - Apply **R2**:
    - Apply **R4**: gets the number ("3")
    - Expects nothing, but gets a "*": backtrack

A simple grammar:

(R1) Expr ➔ Number + Expr

(R2) Expr ➔ Number

(R3) Expr ➔ Number * Expr

(R4) Number ➔ 0-9

Top-Down Parsing Example

- Let’s parse string “3*4*5”
  - Apply **R1**:
    - Apply **R4**: gets the number (“3”)
    - Expects a “+”, but gets a “*”: backtrack
  - Apply **R2**:
    - Apply **R4**: gets the number (“3”)
    - Expects nothing, but gets a “*”: backtrack
  - Apply **R3**:
    - Apply **R4**: gets the number (“3”)
    - Gets the “*”
    - Apply **R1**
      - Apply **R4**: gets the number (“4”)
      - Expects a “+”, but gets a “*”: backtrack

A simple grammar:

(R1) Expr ➔ Number + Expr

(R2) Expr ➔ Number

(R3) Expr ➔ Number * Expr

(R4) Number ➔ 0-9

Top-Down Parsing Example

- Let's parse string "3*4*5"
  - Apply **R1**:
    - Apply **R4**: gets the number ("3")
    - Expects a "+", but gets a "*": backtrack
  - Apply **R2**:
    - Apply **R4**: gets the number ("3")
    - Expects nothing, but gets a "*": backtrack
  - Apply **R3**:
    - Apply **R4**: gets the number ("3")
    - Gets the "*"
    - Apply **R1**
      - Apply **R4**: gets the number ("4")
      - Expects a "+", but gets a "*": backtrack
    - Apply **R2**
      - Apply **R4**: gets the number ("4")
      - Expects nothing, but gets a "*": backtrack

A simple grammar:

(R1) Expr ➔ Number + Expr

(R2) Expr ➔ Number

(R3) Expr ➔ Number * Expr

(R4) Number ➔ 0-9

Top-Down Parsing Example

- Let's parse string "3*4*5"
  - Apply **R1**:
    - Apply **R4**: gets the number ("3")
    - Expects a "+", but gets a "*": backtrack
  - Apply **R2**:
    - Apply **R4**: gets the number ("3")
    - Expects nothing, but gets a "*": backtrack
  - Apply **R3**:
    - Apply **R4**: gets the number ("3")
    - Gets the "*"
    - Apply **R1**
      - Apply **R4**: gets the number ("4")
      - Expects a "+", but gets a "*": backtrack
    - Apply **R2**
      - Apply **R4**: gets the number ("4")
      - Expects nothing, but gets a "*": backtrack
    - Apply **R3**:
      - Apply **R4**: gets the number ("4")
      - Gets the "*"
      - Apply **R1**:
        - Apply **R4**: gets the number ("5")
        - Expects a "+", but gets nothing: backtrack
      - Apply **R2**:
        - Apply **R4**: gets the number ("5"): done

A simple grammar:

(R1) Expr ➔ Number + Expr

(R2) Expr ➔ Number

(R3) Expr ➔ Number * Expr

(R4) Number ➔ 0-9

Left-Recursion

- One problem for the Top-Down approach is left-recursive rules
- Example: Expr ➔ Expr + Number
  - The Parser will expand the leftmost Expr as Expr + Number to get: “Expr + Number + Number”
  - And again: “Expr + Number + Number + Number”
  - And again: “Expr + Number + Number + Number + Number”
  - Ad infinitum. . .
  - Since the leftmost symbol is never a non-terminal symbol the parser will never check for a match with the source code and will be stuck in an infinite loop
- Luckily, there are ways to remove left-recursion

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down and is the method typically used in practice
- The idea is very simple:
  - Look at the string of tokens, from left to right
  - Look for “things that look like” the right-hand side of production rules
  - Replace the tokens
- Intuitively, it seems less “random” than Top-Down parsing and more “clever”
- Let’s see an example

Bottom-Up Parsing Example

- The same simple grammar

(R1) Expr ➔ Expr * Number

(R2) Expr ➔ Number

(R3) Expr ➔ Number * Number

(R4) Number ➔ 0-9

- Let’s parse string “3*4*5”
  - Number * 4 * 5 (R4)
  - Number * Number * 5 (R4)
  - Expr * 5 (R3)
  - Expr * Number (R4)
  - Expr (R5) [done]

Bottom-Up Parsing

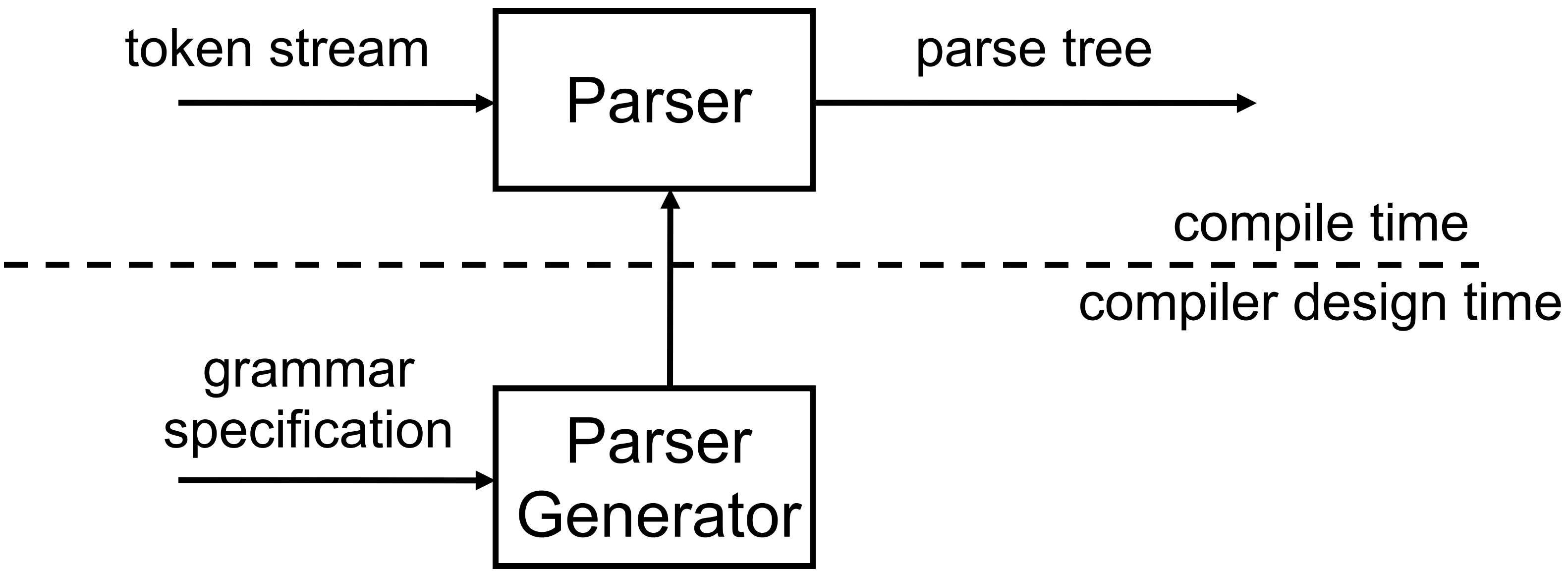
- The previous example made it look very simple, but this doesn't always work
- Turns out there is a way to do this ("shift-reduce" parsing) that is guaranteed to work for any non-ambiguous grammar
  - Uses a stack to do some backtracking
- More about all this in a graduate-level compiler course…

Parsing: Bottom-Line

- Parsing was a very active field of research and development decades ago
- At this point it's mostly well-understood
  - We know what properties grammars should have to have easy/quick parsers
  - We know which parsing algorithms work well

- As a result, we have tools that generate the parser code for us…

Parser Generator



So What Now?

- We want to write a compiler for a given language
- Lexing
  - We come up with a definition of the tokens embodied in regular expressions
  - We build a lexer using a tool
  - In the previous set of lecture notes, we have used ANTLR to do this
- Parsing
  - We come up with a definition of the syntax embodied in a context-free grammar
  - We build a parser using a tool
  - Let’s use ANTLR again for a simple language!

Our Language

- We have all the tokens we've already defined in our lexer:
  - IF, ENDIF
  - PRINT, INT, PLUS, LPAREN, RPAREN
  - EQUAL, NOTEQUAL, ASSIGN, SEMICOLON
  - INTEGER, NAME
- We want a very limited language with
  - integer variable declarations
  - assignments
  - addition (only 2 operands)
  - if (not else, only test for equality)
  - semicolon-terminated statements
  - white-spaces, tabs, carriage returns don't matter
- Let's look at an example program to get a sense of it

Example Program

```
int a;
int b;
a = 3;
b = a + 1;
if (b != 4)
    a = 2;
endif
if (a == 3)
     a = a + 1;
     b = b + 6;
endif
print a;
print b;
```

Let’s Implement the Parser

- Let’s attempt live-coding of the Parser for our language right now using ANTLR…
- The basic ANTLR syntax for a production rule looks like this:

```
expression      :
        expression  PLUS expression |
        expression MINUS expression
                ;

number          :
        digit |
        digit *
                ;
```

Did we succeed?

- I had done this parser beforehand, and it is posted on the Web site
  - In the “A Simple ANTLR Parser” reading

- We’ll use the one of the Web site for the next step

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 our compiler generate code
- In the next set of lecture notes we will make our compiler generate x86 assembly!!!