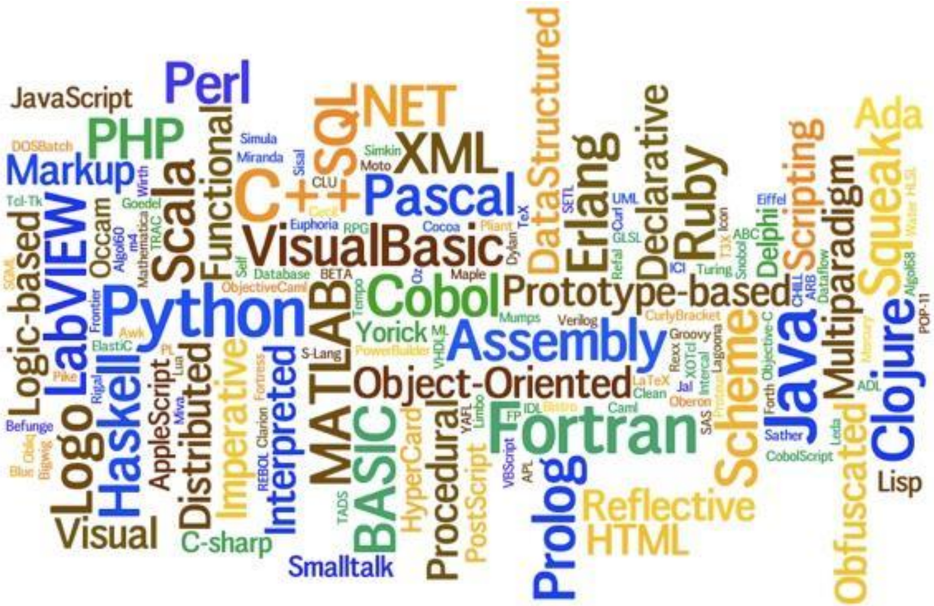


Programming Languages Syntax II

Theory and fundamentals of Programming Languages

Module 3

Dr. Tamer ABUHMED

SUNG KYUN KWAN UNIVERSITY

Outline

- The General Problem of Describing Syntax Cont.
  - Formal Methods of Describing Syntax
- Operators: Precedence and Associativity
- Syntactic Sugar
- Extended BNF
- Parsing Complexity

Recap: An ambiguous grammar

- Here is a simple grammar for expressions that is ambiguous
  - <e> → <e> <op> <e>
  - <e> → 1 | 2 | 3
  - <op> → + | - | * | /

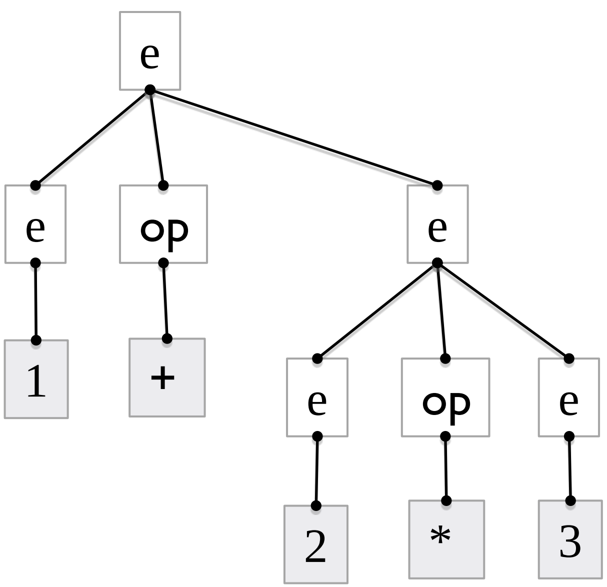
Fyi… In a programming language, an expression is some code that is evaluated and produces a value. A statement is code that is executed and does something but does not produce a value.

- The sentence *1+2*3* can lead to two different parse trees corresponding to *1+(2*3)* and *(1+2)*3*

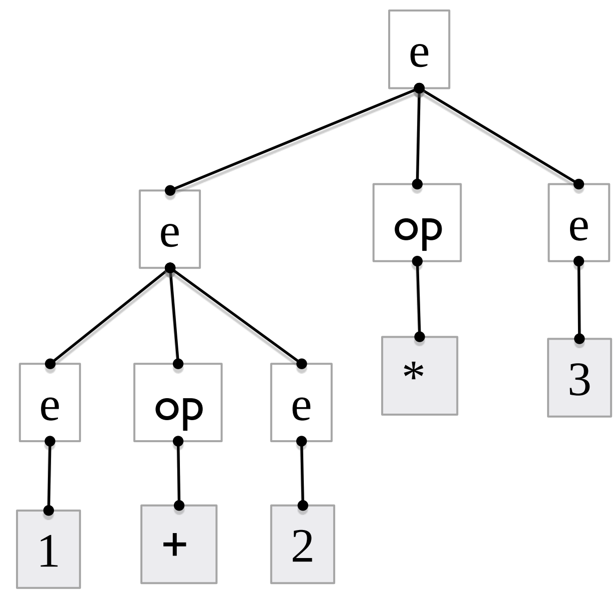
Two derivations for 1+2*3

```
<e> → <e> <op> <e>
<e> → 1|2|3
<op> → +|-|*|/
```

```
<e> → <e> <op> <e>
      → 1 <op> <e>
      → 1 + <e>
     → 1 + <e> <op> <e>
     → 1 + 2 <op> <e>
  → 1 + 2 * <e>
  → 1 + 2 * 3
```




```
<e> → <e> <op> <e>
 → <e> <op> <e> <op> <e>
 → 1 <op> <e> <op> <e>
 →  1 + <e> <op> <e>
 →  1 + 2 <op> <e>
 → 1 + 2 * <e>
 → 1 + 2 * 3
```



The leaves of the trees are terminals and correspond to the sentence

Operators

- The traditional operator notation introduces many problems.
- Operators are used in
  - Prefix notation: Expression *(* (+ 1 3) 2)* in Lisp
  - Infix notation: Expression *(1 + 3)* * 2 in Java
  - Postfix notation: Increment *foo*++ in C
- Operators can have one or more operands
  - Increment in C is a one-operand operator: *foo*++
  - Subtraction in C is a two-operand operator: *foo - bar*
  - Conditional expression in C is a three-operand operators: *(foo == 3 ? 0 : 1)*

Resolve ambiguous grammar

- Example:

<e> $\rightarrow$ <e> + <e>

<e> $\rightarrow$ <e> * <e>

<e> $\rightarrow$ id

Operator notation

- So, how do we interpret expressions like

  (a) 2 + 3 + 4

  (b) 2 + 3 * 4
- While you might argue that it doesn't matter for (a), it can for different operators *(2 ** 3 ** 4)* or when the limits of representation are hit (e.g., round off in numbers, e.g., 1+1+1+1+1+1+1+1+1+1+1+10**6)
- Concepts:
  - Explaining rules in terms of operator precedence and associativity
  - Realizing the rules in grammars

Operators: Precedence and Associativity

- Precedence and associativity deal with the evaluation order within expressions
- *Precedence* rules specify order in which operators of different precedence level are evaluated, e.g.:

  "*" Has a higher precedence that "+", so "*" groups *more tightly* than "+"
- What is the results of 4 * 5 ** 6 ?
- A language's precedence hierarchy should match our intuitions, but the result's not always perfect, as in this Pascal example:

  if A<B and C<D then A := 0 ;
- Pascal relational operators have lowest precedence!

  if A < B and C < D then A := 0 ;

Operator Precedence: Precedence Table

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), & (address of), * (contents of), ! (logical not), ~ (bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, -	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)		<, >, <=, >= (inequality tests)	=, /=, <=, >, >= (comparisons)
.not.		==, != (equality tests)	

Operator Precedence: Precedence Table

	& (bit-wise and)	
	^ (bit-wise exclusive or)	
	| (bit-wise inclusive or)	
.and.	&& (logical and)	and, or, xor (logical operators)
.or.	|| (logical or)	
.eqv., .neqv. (logical comparisons)	?: (if...then...else)	
	=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
	, (sequencing)	

Operators: Associativity

- *Associativity* rules specify order in which operators of the same precedence level are evaluated
- Operators are typically either left associative or right associative.
- Left associativity is typical for +, - , * and /
- So A + B + C
  - Means: (A + B) + C
  - And not: A + (B + C)
- Does it matter?

Operators: Associativity

- For + and * it doesn't matter in theory (though it can in practice) but for – and / it matters in theory, too.
- What should A-B-C mean?

  (A – B) – C ≠ A – (B – C)
- What is the results of 2 ** 3 ** 4 ?
  - 2 ** (3 ** 4) = 2 ** 81 = 2417851639229258349412352
  - (2 ** 3) ** 4 = 8 ** 4 = 256
- Languages diverge on this case:
  - In Fortran, ** associates from right-to-left, as in normally the case for mathematics
  - In Ada, ** doesn't associate; you must write the previous expression as 2 ** (3 ** 4) to obtain the expected answer

Associativity in C

- In C, as in most languages, most of the operators associate left to right

  a + b + c => (a + b) + c

- The various assignment operators however associate right to left

  = += -= *= /= %= >>= <<= &= ^= |=

- Consider a += b += c, which is interpreted as

  a += (b += c)

- and not as

  (a += b) += c

- Why?

Precedence and associativity in Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity
- An unambiguous expression grammar:
  - `<expr> -> <expr> - <term>  |  <term>`
  - `<term> -> <term> / const  |  const`
-

Precedence and associativity in Grammar

Sentence: const – const / const

Derivation:

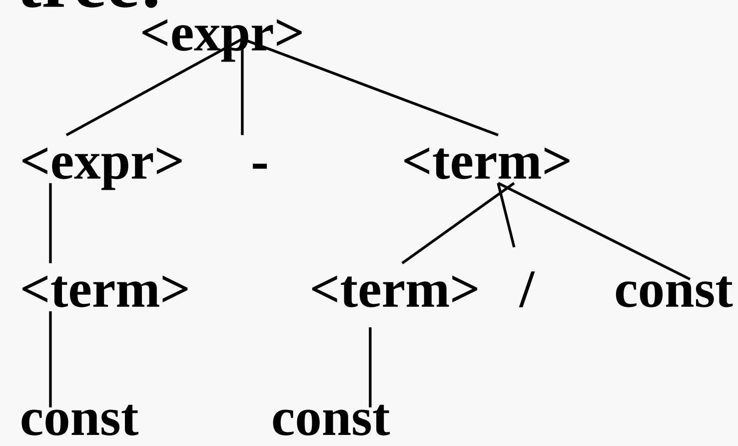
<expr> => <expr> **-** <term>
=> <term> **-** <term>
=> const **-** <term>
=> const **-** <term> / const
=> const **-** const / const

Grammar

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

<term> -> <term> /
const  |  const
```

Parse tree:

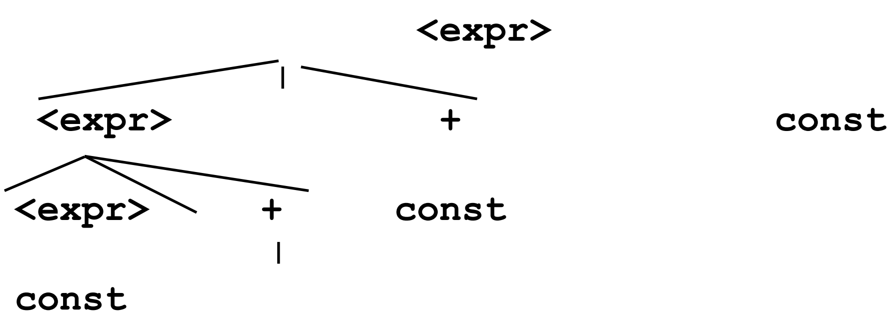


Grammar (continued)

Operator associativity can also be indicated by a grammar

<expr> -> <expr> + <expr> | const (ambiguous)

<expr> -> <expr> + const | const (unambiguous)



Does this grammar rule make the + operator right or left associative?

An Expression Grammar

Here's a grammar to define simple arithmetic expressions over variables and numbers.

Exp ::= num
Exp ::= id
Exp ::= UnOp Exp
Exp := Exp BinOp Exp
Exp ::= '(' Exp ')'

UnOp ::= '+'
UnOp ::= '-'
BinOp ::= '+' | '-' | '*' | '/'

Here's another common notation variant where single quotes are used to indicate terminal symbols and unquoted symbols are taken as non-terminals.

A derivation

A derivation of a+b*2 using the expression grammar:

Exp => // Exp ::= Exp BinOp Exp

Exp BinOp Exp => // Exp ::= id

id BinOp Exp => // BinOp ::= '+'

id + Exp => // Exp ::= Exp BinOp Exp

id + Exp BinOp Exp => // Exp ::= num

id + Exp BinOp num => // Exp ::= id

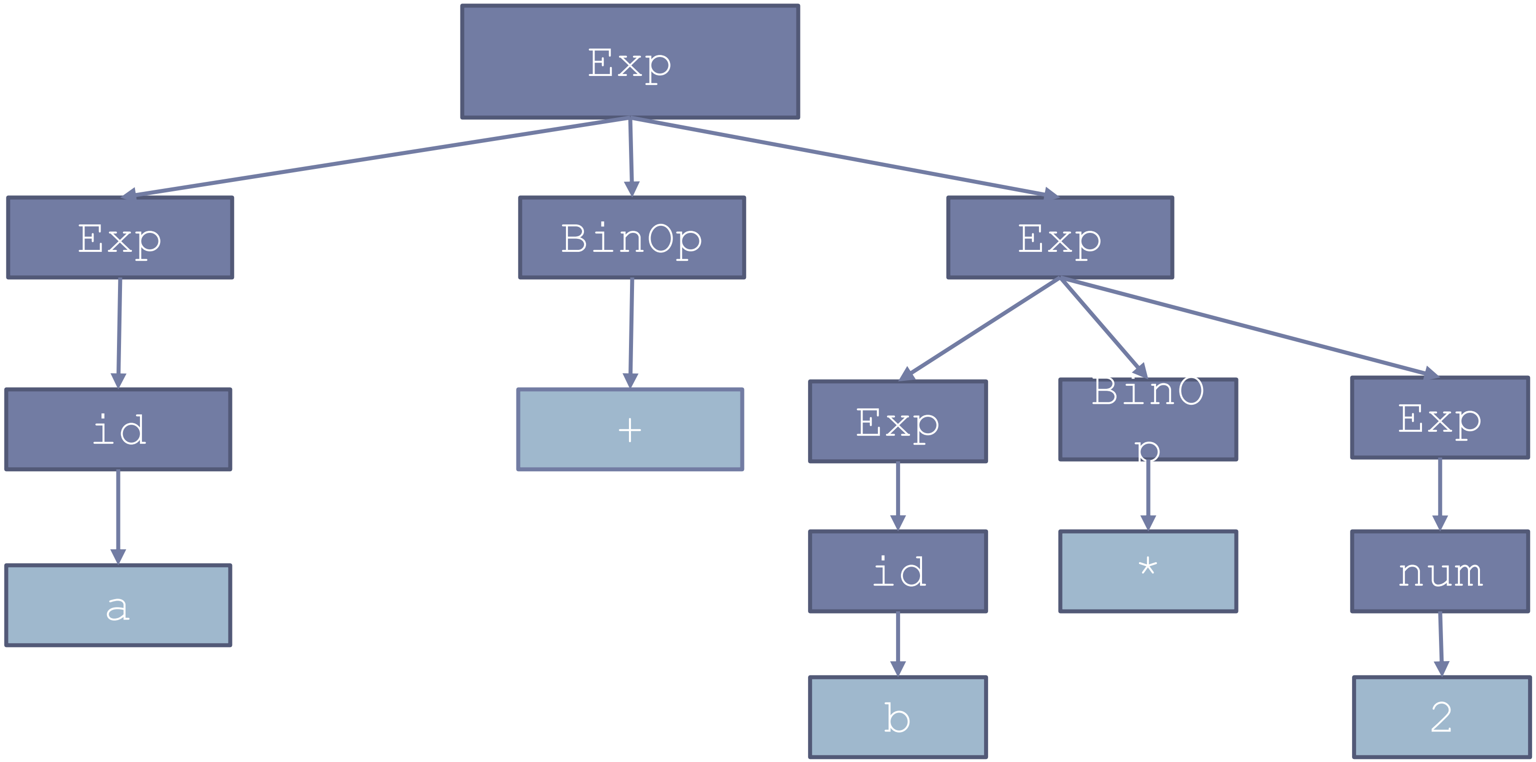
id + id BinOp num => // BinOp ::= '*'

id + id * num

a + b * 2

A parse tree

A parse tree for a+b*2:



Precedence

- Precedence refers to the order in which operations are evaluated
- Usual convention: exponents > mult, div > add, sub
- Deal with operations in categories: exponents, mulops, addops.
- A revised grammar that follows these conventions:

```
Exp ::= Exp AddOp Exp
Exp ::= Term
Term ::= Term MulOp Term
Term ::= Factor
Factor ::= '(' + Exp + ')‘
Factor ::= num | id
AddOp ::= '+' | '-’
MulOp ::= '*' | '/'
```

Associativity

- Associativity refers to the order in which two of the same operation should be computed
  - 3+4+5 = (3+4)+5, left associative (all BinOps)
  - 3^4^5 = 3^(4^5), right associative
- Conditionals right associate but have a wrinkle: an else clause associates with closest *unmatched if*

  if a then if b then c else d

  = if a then (if b then c else d)

Adding associativity to the grammar

Adding associativity to the BinOp expression grammar

Exp ::= Exp AddOp Term

Exp ::= Term

Term ::= Term MulOp Factor

Term ::= Factor

Factor ::= '(' Exp ')'

Factor ::= num | id

AddOp ::= '+' | '-'

MulOp ::= '*' | '/'

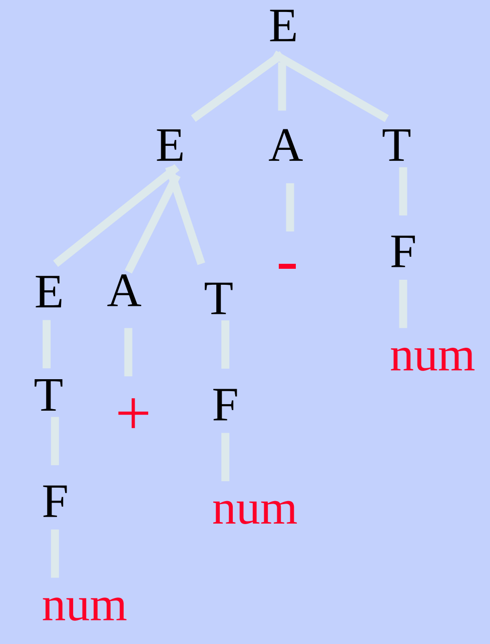
Grammar

```
Exp    ::= Exp AddOp Term
Exp    ::= Term
Term   ::= Term MulOp Factor
Term   ::= Factor
Factor ::= '(' Exp ')'
Factor ::= num | id
AddOp  ::= '+' | '-'
MulOp  ::= '*' | '/'
```

Derivation

```
Exp =>
Exp AddOp Term =>
Exp AddOp Exp AddOp Term =>
Term AddOp Exp AddOp Term =>
Factor AddOp Exp AddOp Term =>
Num AddOp Exp AddOp Term =>
Num + Exp AddOp Term =>
Num + Factor AddOp Term =>
Num + Num AddOp Term =>
Num + Num - Term =>
Num + Num - Factor =>
Num + Num - Num
```

Parse tree



Example: conditionals

- Most languages allow two conditional forms, with and without an else clause:
  - if x < 0 then x = -x
  - if x < 0 then x = -x else x = x+1
- But we'll need to decide how to interpret:
  - if x < 0 then if y < 0 x = -1 else x = -2
- To which if does the else clause attach?
- This is like the syntactic ambiguity in attachment of prepositional phrases in English
  - the man near a cat with a hat

Example: conditionals

- All languages use standard rule to determine which if expression an else clause attaches to
- The rule:
  - An else clause attaches to the nearest if to its left that does not yet have an else clause
- Example:
  - if x < 0 then if y < 0 x = -1 else x = -2
  - if x < 0 then <u>if y < 0 x = -1 else x = -2</u>

Example: conditionals

- Goal: to create a correct grammar for conditionals.
- It needs to be non-ambiguous and the precedence is else with nearest unmatched if

Statement ::= Conditional | 'whatever'

Conditional ::= 'if' test 'then' Statement 'else‘ Statement

Conditional ::= 'if' test 'then' Statement

- The grammar is ambiguous. The first Conditional allows unmatched ifs to be Conditionals
  - Good: if test then (if test then whatever else whatever)
  - Bad: if test then (if test then whatever) else whatever
- Goal: write a grammar that forces an else clause to attach to the nearest if w/o an else clause

Example: conditionals

The final unambiguous grammar

```
Statement ::= Matched | Unmatched
Matched ::= 'if' test 'then' Matched 'else' Matched
             | 'whatever'
Unmatched ::= 'if' test 'then' Statement
             | 'if' test 'then' Matched ‘else’ Unmatched
```

Syntactic Sugar

- Syntactic sugar: syntactic features designed to make code easier to read or write while alternatives exist
- Makes the language *sweeter* for humans to use: things can be expressed more clearly, concisely, or in an alternative style that some prefer
- Syntactic sugar can be removed from language without effecting what can be done
- All applications of the construct can be systematically replaced with equivalents that don't use it

adapted from Wikipedia

Syntactic Sugar: Python example

```python
Full_List = [(1, 0), (2, 1), (3, 5), (4, 7), (5, 5)]
filter = [1, 3]


#The ugly
new_list = []
for id, count in Full_List:
    if id not in filter:
        new_list.append((id, count))
print (new_list)
new_list = []
#The Pythonic way
new_list = [ (id, c) for id, c in Full_List\
             if id not in filter]
print (new_list)
```

Extended BNF

- *Syntactic sugar:* doesn’t extend the expressive power of the formalism, but does make it easier to use, i.e., more readable and more writable
- Optional parts are placed in brackets ([])
- <proc_call> -> ident [(<expr_list>)]
- Put alternative parts of RHSs in parentheses and separate them with vertical bars
- <term> -> <term> (+ | -) const
- Put repetitions (0 or more) in braces ({})
- <ident> -> letter {letter | digit}

BNF vs EBNF

- BNF:

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

- EBNF:

```
<expr> -> <term> {(+ | -) <term>}
<term> -> <factor> {(* | /) <factor>}
```

Parsing

- A grammar describes the strings of tokens that are syntactically legal in a PL
- A *recogniser* simply accepts or rejects strings.
- A generator produces sentences in the language described by the grammar
- A *parser* construct a derivation or parse tree for a sentence (if possible)
- Two common types of parsers are:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A *recursive descent parser* is a way to implement a top-down parser that is particularly simple.

A Bottom-up Parse in Example (1)

$E \rightarrow int$
$E \rightarrow E + (E)$

int + (int) + (int)

int + (int) + (int)

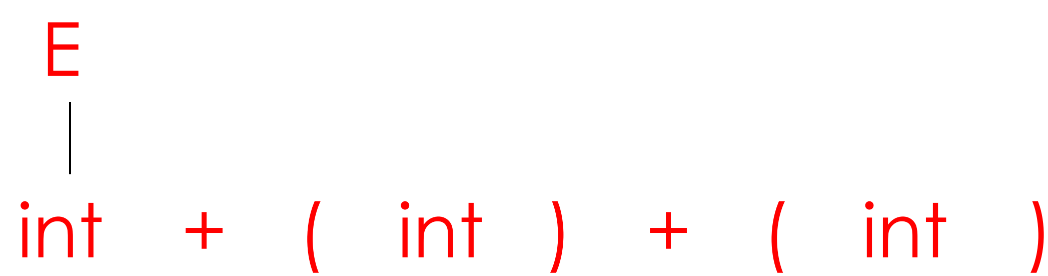
A Bottom-up Parse in Example (2)

int + (int) + (int)

E + (int) + (int)

$E \rightarrow int$

$E \rightarrow E + (E)$



A Bottom-up Parse in Example (3)

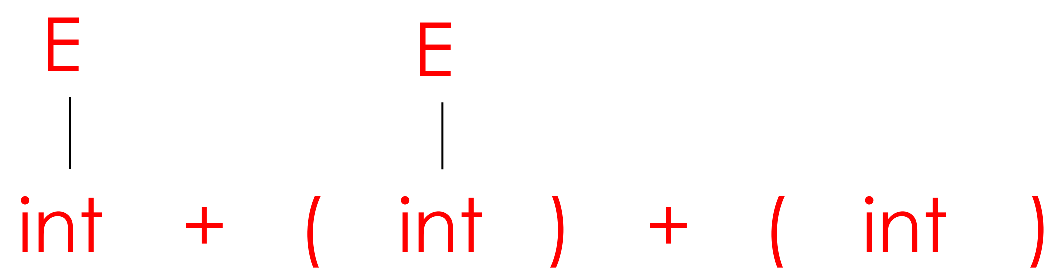
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

$E \rightarrow int$

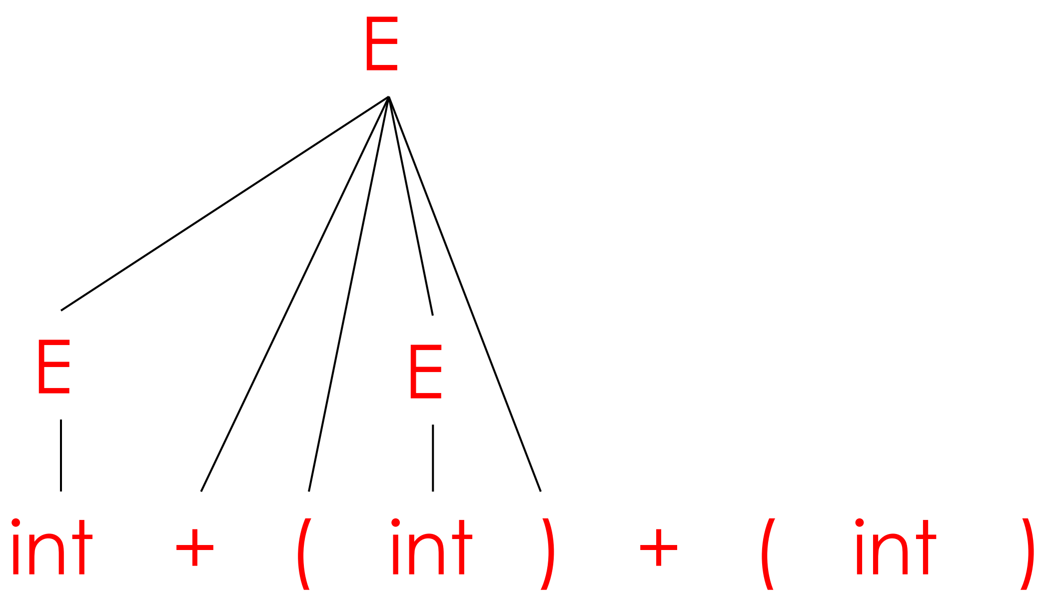
$E \rightarrow E + (E)$



A Bottom-up Parse in Example (4)

$E \rightarrow$ int
$E \rightarrow E$ + (E)

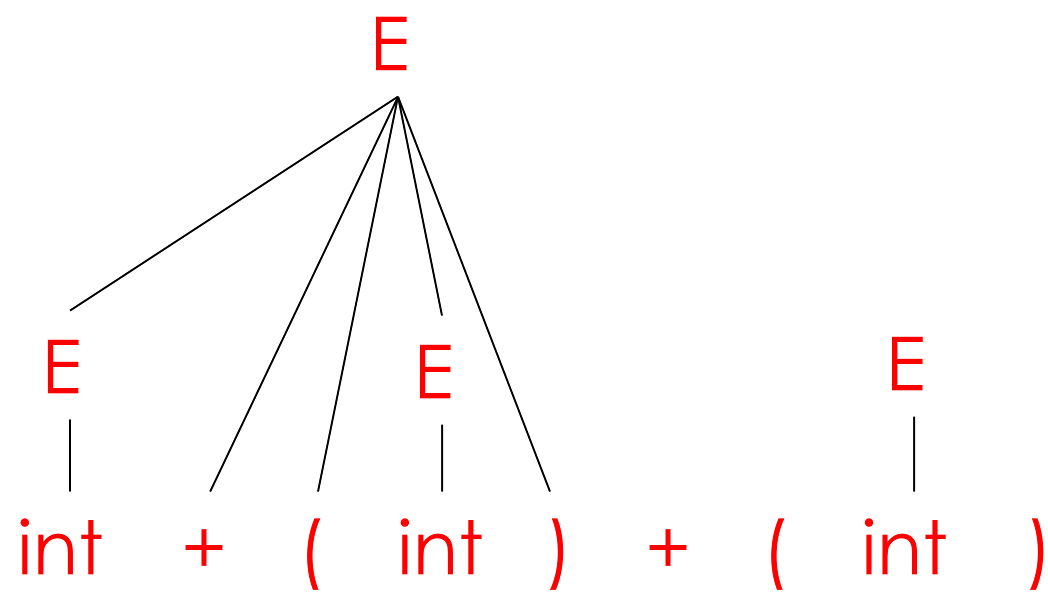
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)



A Bottom-up Parse in Example (5)

E → int
E → E + (E)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)



A Bottom-up Parse in Example (6)

int + (int) + (int)

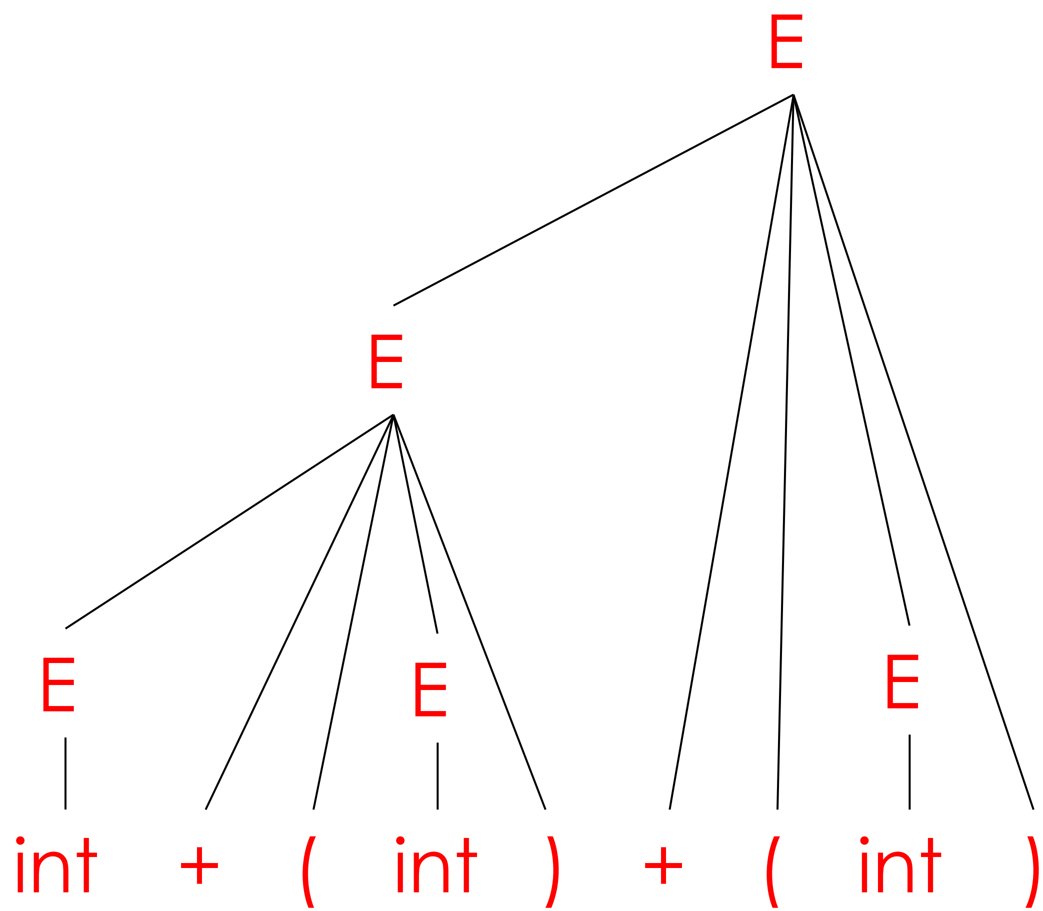
E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)

E



E → int

E → E + (E)

Parsing complexity

- How hard is the parsing task?
- Parsing an arbitrary context free grammar is $O(n^3)$, e.g., it can take time proportional the cube of the number of symbols in the input. This is bad!
- If we constrain the grammar somewhat, we can always parse in linear time. This is good!
- Linear-time parsing
  - LL parsers
    - » Recognize LL grammar
    - » Use a top-down strategy
  - LR parsers
    - » Recognize LR grammar
    - » Use a bottom-up strategy

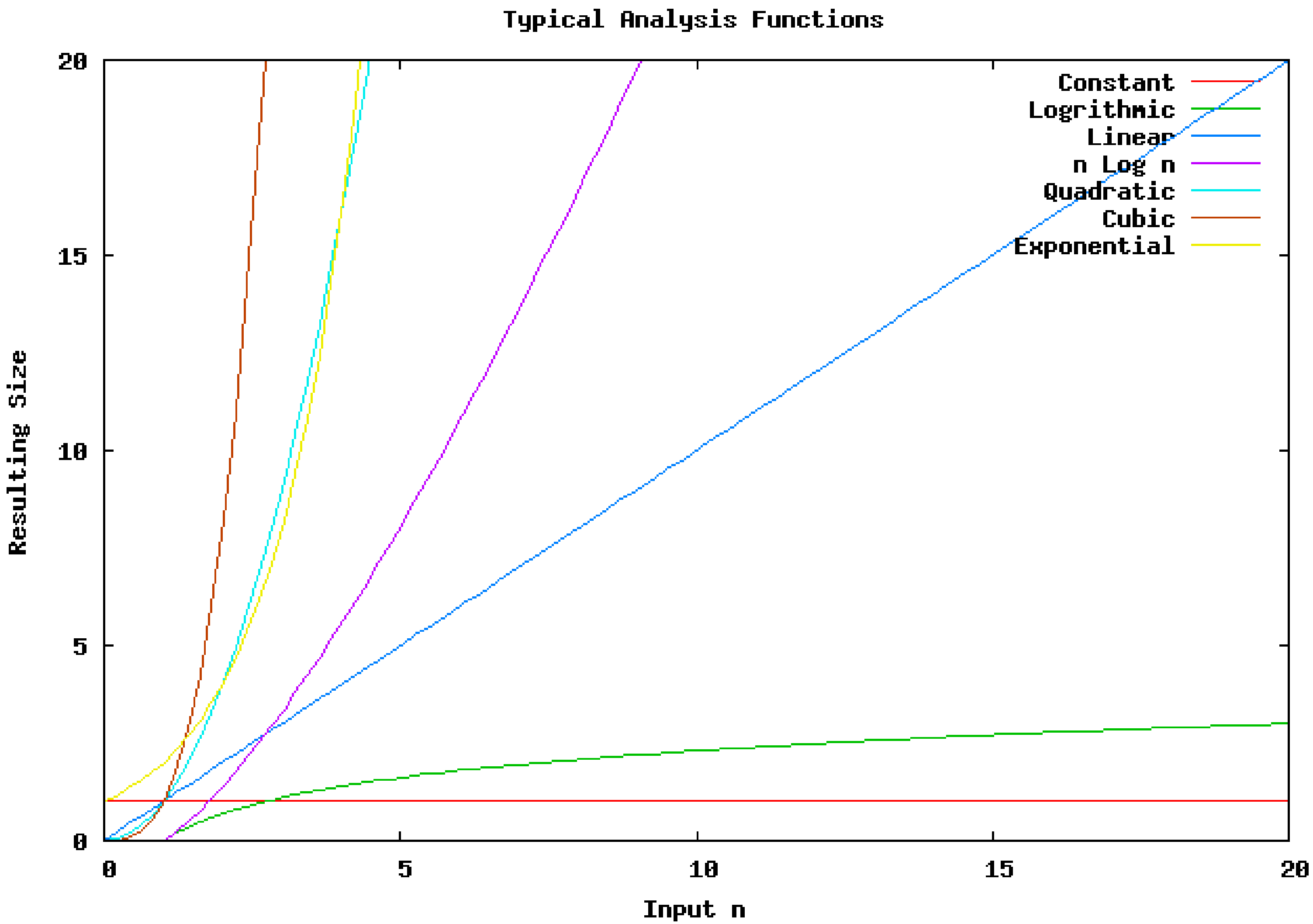
- LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.
- LR(n) : Left to right, Right derivation, look ahead at most n symbols.

Parsing complexity

- If it takes t_1 seconds to parse your C program with n lines of code, how long will it take to take if you make it twice as long?
  - time(n) = $t_{1,}$ time(2n) = 2^3 * time(n)
  - 8 times longer
- Suppose v3 of your code is has 10n lines?
  - 10^3 or 1000 times as long
- Windows Vista was said to have ~50M lines of code

- Practical parsers have time complexity that is linear in the number of tokens, i.e., O(n)
- If your program is twice as long, it will take twice as long to parse

Parsing complexity



Summary

- The syntax of a programming language is usually defined using BNF or a context free grammar
- In addition to defining what programs are syntactically legal, a grammar also encodes meaningful or useful abstractions (e.g., block of statements)
- Typical syntactic notions like operator precedence, associativity, sequences, optional statements, etc. can be encoded in grammars
- A parser is based on a grammar and takes an input string, does a derivation and produces a parse tree.