3 Syntax
Some Preliminaries

• For the next several weeks we’ll look at how one can define a programming language

• What is a language, anyway?
  “Language is a system of gestures, grammar, signs, sounds, symbols, or words, which is used to represent and communicate concepts, ideas, meanings, and thoughts”

• Human language is a way to communicate representations from one (human) mind to another

• What about a programming language?
  A way to communicate representations (e.g., of data or a procedure) between human minds and/or machines
**Introduction**

We usually break down the problem of *defining* a programming language into two parts

- defining the PL’s **syntax**
- defining the PL’s **semantics**

*Syntax* - the **form** or structure of the expressions, statements, and program units

*Semantics* - the **meaning** of the expressions, statements, and program units

Note: There is not always a clear boundary between the two
Why and How

Why? We want specifications for several communities:
  • Other language designers
  • Implementers
  • Machines?
  • Programmers (the users of the language)

How? One ways is via natural language descriptions (e.g., user’s manuals, text books) but there are a number of techniques for specifying the syntax and semantics that are more formal.
This is an overview of the standard process of turning a text file into an executable program.
Syntax Overview

• Language preliminaries
• Context-free grammars and BNF
• Syntax diagrams
Introduction

A *sentence* is a string of characters over some alphabet (e.g., `def add1(n): return n + 1`)

A *language* is a set of sentences

A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `add1`, `begin`)

A *token* is a category of lexemes (e.g., `identifier`)

Formal approaches to describing syntax:

- Recognizers - used in compilers
- Generators - what we'll study
Lexical Structure of Programming Languages

- The structure of its lexemes (words or tokens)
  - token is a category of lexeme
- The scanning phase (lexical analyser) collects characters into tokens
- Parsing phase (syntactic analyser) determines syntactic structure
Grammars

Context-Free Grammars

• Developed by Noam Chomsky in the mid-1950s. (View Chomsky vs. Bill Buckley on YouTube)
• Language generators, meant to describe the syntax of natural languages.
• Define a class of languages called context-free languages.

Backus Normal/Naur Form (1959)

• Invented by John Backus to describe Algol 58 and refined by Peter Naur for Algol 60.
• BNF is equivalent to context-free grammars
Chomsky & Backus independently came up with equivalent formalisms for specifying the syntax of a language.
Backus focused on a practical way of specifying an artificial language, like Algol.
Chomsky made fundamental contributions to mathematical linguistics and was motivated by the study of human languages.

A metalanguage is a language used to describe another language.

In BNF, abstractions are used to represent classes of syntactic structures -- they act like syntactic variables (also called nonterminal symbols), e.g.

```
<while_stmt> ::= while <logic_expr> do <stmt>
```

This is a rule; it describes the structure of a while statement. Often the word production is used for rule.
BNF

• A rule has a left-hand side (LHS) which is a single non-terminal symbol and a right-hand side (RHS), one or more terminal or non-terminal symbols.
• A grammar is a finite, nonempty set of rules.
• A non-terminal symbol is “defined” by its rules.
• Multiple rules can be combined with the vertical-bar ( | ) symbol (read as “or”)
• These two rules:

  \[
  \langle \text{stmts} \rangle ::= \langle \text{stmt} \rangle \\
  \langle \text{stmts} \rangle ::= \langle \text{stmt} \rangle ; \langle \text{stmts} \rangle
  \]

are equivalent to this one:

  \[
  \langle \text{stmts} \rangle ::= \langle \text{stmt} \rangle | \langle \text{stmt} \rangle ; \langle \text{stmts} \rangle
  \]
Non-terminals, pre-terminals & terminals

• A **non-terminal** symbol is any symbol that is on the LHS of a rule. These represent abstractions in the language (e.g., *if-then-else-statement* in
  
  `<if-then-else-statement> ::= if <test> then <statement> else <statement>`

• A **terminal** symbol is any symbol that is not on the LHS of any rule. AKA *lexemes*. These are the literal symbols that will appear in a program (e.g., *if*, *then*, *else* in rules above).

• A **pre-terminal** symbol is a non-terminal that appears on the LHS of >= 1 rule(s), but in every case, the RHSs consist of single terminal symbols, e.g., `<digit>` in
  
  `<digit> ::= 0 | 1 | 2 | 3 ... 7 | 8 | 9`
BNF

- Repetition is done with recursion
- An `<ident_list>` is a sequence of one or more `<ident>`s separated by commas.

```
<ident_list> ::= <ident> | <ident> , <ident_list>
```
BNF Example

Here is an example of a simple grammar for a small subset of English

A sentence is a noun phrase and verb phrase followed by a period.

\[
\begin{align*}
\text{<sentence>} & \ ::= \text{<nounPhrase>} \ \text{<verbPhrase>} \ . \\
\text{<nounPhrase>} & \ ::= \text{<article>} \ \text{<noun>} \\
\text{<article>} & \ ::= \text{a} \ | \ \text{the} \\
\text{<noun>} & \ ::= \text{man} \ | \ \text{apple} \ | \ \text{worm} \ | \ \text{penguin} \\
\text{<verbPhrase>} & \ ::= \text{<verb>} | \text{<verb><nounPhrase>} \\
\text{<verb>} & \ ::= \text{eats} \ | \ \text{throws} \ | \ \text{sees} \ | \ \text{is}
\end{align*}
\]
Derivations

• A *derivation* is a repeated application of rules, starting with the start symbol and ending with a sentence consisting only of terminal symbols.

• It demonstrates, or proves that the derived sentence is “generated” by the grammar and is thus in the language that the grammar defines.

• As an example, consider our baby English grammar:

  `<sentence>    ::= <nounPhrase><verbPhrase>`.
  `<nounPhrase> ::= <article><noun>`
  `<article> ::= a | the`
  `<noun> ::= man | apple | worm | penguin`
  `<verbPhrase> ::= <verb> | <verb><nounPhrase>`
  `<verb> ::= eats | throws | sees | is`
Derivation using BNF

Here is a derivation for “the man eats the apple.”

\[ \text{<sentence>} \rightarrow \text{<nounPhrase>;<verbPhrase>}. \]
\[ \text{<article>;<noun>;<verbPhrase>}. \]
\[ \text{the;<noun>;<verbPhrase>}. \]
\[ \text{the man;<verbPhrase>}. \]
\[ \text{the man;<verb>;<nounPhrase>}. \]
\[ \text{the man eats;<nounPhrase>}. \]
\[ \text{the man eats;<article>;<noun>}. \]
\[ \text{the man eats the;<noun>}. \]
\[ \text{the man eats the apple}. \]
Derivation

Every string of symbols in the derivation is a *sentential form*.

A *sentence* is a sentential form that has only terminal symbols.

A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded in the next step.

A derivation may be either leftmost or rightmost or something else.
Another BNF Example

\[
\langle \text{program} \rangle \rightarrow \langle \text{stmts} \rangle \\
\langle \text{stmts} \rangle \rightarrow \langle \text{stmt} \rangle \\
\quad \quad \quad | \quad \langle \text{stmt} \rangle ; \langle \text{stmts} \rangle \\
\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \\
\langle \text{var} \rangle \rightarrow a \mid b \mid c \mid d \\
\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle - \langle \text{term} \rangle \\
\langle \text{term} \rangle \rightarrow \langle \text{var} \rangle \mid \text{const} \\
\]

Here is a derivation:

\[
\langle \text{program} \rangle \Rightarrow \langle \text{stmts} \rangle \\
\Rightarrow \langle \text{stmt} \rangle \\
\Rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \\
\Rightarrow a = \langle \text{expr} \rangle \\
\Rightarrow a = \langle \text{term} \rangle + \langle \text{term} \rangle \\
\Rightarrow a = \langle \text{var} \rangle + \langle \text{term} \rangle \\
\Rightarrow a = b + \langle \text{term} \rangle \\
\Rightarrow a = b + \text{const} \\
\]

Note: There is some variation in notation for BNF grammars. Here we are using \(\rightarrow\) in the rules instead of \(::=\).
Finite and Infinite languages

• A simple language may have a finite number of sentences

• An example of a finite language is the set of strings representing integers between -10**6 and +10**6

• A finite language can be defined by enumerating the sentences, but using a grammar might be much easier

• Most interesting languages have an infinite number of sentences (why?)
Is English a finite or infinite language?

• Assume we have a finite set of words
• Consider adding rules like the following to the previous example
  \[ <\text{sentence}> ::= <\text{sentence}><\text{conj}><\text{sentence}>. \]
  \[ <\text{conj}> ::= \text{and} | \text{or} | \text{because} \]
• Hint: Whenever you see recursion in a BNF it’s likely that the language is infinite.
  – When might it not be?
A parse tree is a hierarchical representation of a derivation.

Parse Tree: A parse tree for the expression `a + b` const.
Another Parse Tree

The man eats the apple.
A grammar is ambiguous if and only if (iff) it generates a sentential form that has two or more distinct parse trees.

Ambiguous grammars are, in general, very undesirable in formal languages.

We can sometimes eliminate ambiguity by changing the grammar.
Ambiguous English Sentences

• I saw the man on the hill with a telescope
• Time flies like an arrow
• Fruit flies like a banana
• Buffalo Buffalo buffalo
  • verb, city, or beast?

See: Syntactic Ambiguity
An ambiguous grammar

Here is an ambiguous grammar for expressions

\[
\begin{align*}
\langle e \rangle & \rightarrow \langle e \rangle \ \langle op \rangle \ \langle e \rangle \\
\langle e \rangle & \rightarrow 1 \mid 2 \mid 3 \\
\langle op \rangle & \rightarrow + \mid - \mid * \mid /
\end{align*}
\]

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

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.
Two parse trees for $1+2*3$

$<e> \rightarrow <e> <op> <e>$
$<e> \rightarrow 1|2|3$
$<op> \rightarrow +|-|*|/$
Operators

• The traditional operator notation introduces many problems.

• Operators are used in
  – Prefix notation: Expression \((\ast (\,\, + \,\, 1\,\, 3)\,\, 2)\) 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: \((foo == 3 \,\, ? \,\, 0 \,\, : \,\, 1)\)
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., $1+1+1+1+1+1+1+1+1+1+10**6$)

• 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 than “+”, so “*” groups **more tightly** than “+”.

- What is the value of the expression \( 4 \times 5^2 \times 6 \)?

- A language’s precedence hierarchy should match our intuitions, but the result’s not always perfect, as in this Pascal example:
  
  ```pascal
  if A < B and C < D then A := 0 ;
  ```

- Pascal relational operators have lowest precedence:
  
  ```pascal
  if A < B and C < D then A := 0 ;
  ```
# Operator Precedence: Precedence Table

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>(bit-wise and)</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>(bit-wise exclusive or)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(bit-wise inclusive or)</td>
</tr>
<tr>
<td>.and.</td>
<td>(logical and)</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>(logical and)</td>
<td></td>
</tr>
<tr>
<td>.or.</td>
<td>(logical or)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.eqv.,</td>
<td>(logical comparisons)</td>
<td></td>
</tr>
<tr>
<td>.neqv.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>(if...then...else)</td>
<td></td>
</tr>
<tr>
<td>=, +=, -=, *, /=, %, &gt;&gt;=, &lt;=, &amp;==, ^=,</td>
<td>=</td>
<td>(assignment)</td>
</tr>
</tbody>
</table>
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 /.

• Right associativity is typical for _________?

• 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?
  \[(A – B) – C \neq A – (B – C)\]

• 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 is 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
  
  \[ a + b + c \rightarrow (a + b) + c \]

• The various assignment operators, however, associate right to left

  \[ = \quad += \quad -= \quad *= \quad /= \quad %= \quad >>= \quad <<= \quad &= \quad ^= \quad |= \]

• 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 grammar to indicate precedence levels of the operators, we avoid certain forms of ambiguity.

An unambiguous expression grammar:

\[
<\text{expr}> \rightarrow <\text{expr}> - <\text{term}> \mid <\text{term}>
\]
\[
<\text{term}> \rightarrow <\text{term}> / \text{const} \mid \text{const}
\]
Precedence and Associativity in Grammar

**Sentence:** const – const / const

**Derivation:**

\[
<\text{expr}> \Rightarrow <\text{expr}> - <\text{term}>
\]

\[
\Rightarrow <\text{term}> - <\text{term}>
\]

\[
\Rightarrow \text{const} - <\text{term}>
\]

\[
\Rightarrow \text{const} - <\text{term}> / \text{const}
\]

\[
\Rightarrow \text{const} - \text{const} / \text{const}
\]
Grammar (continued)

Operator associativity can also be indicated by a grammar

\[ <\text{expr}> \rightarrow <\text{expr}> + <\text{expr}> \mid \text{const} \quad \text{(ambiguous)} \]

\[ <\text{expr}> \rightarrow <\text{expr}> + \text{const} \mid \text{const} \quad \text{(unambiguous)} \]

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.

\[
\text{Exp ::= num} \\
\text{Exp ::= id} \\
\text{Exp ::= UnOp Exp} \\
\text{Exp ::= Exp BinOp Exp} \\
\text{Exp ::= '(': Exp ')'} \\
\text{UnOp ::= '+'} \\
\text{UnOp ::= '-'} \\
\text{BinOp ::= '+ | - | * | /}
\]

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

Here’s 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 $a+b*2$:

```
___Exp___
/   |   \
Exp  BinOp  Exp
|     |   /   |
id    +  Exp  BinOp  Exp
|         |     |   |
|         |     |   |
a    id    *    num
|         |
|         |
b    2
```
Precedence

• As we have said, precedence refers to the order in which operations are evaluated.
• Usual convention: exponents > mult div > add sub.
• So, deal with operations in categories: exponents, mulops, addops.
• Here’s a revised grammar that follows these conventions:

\[
\begin{align*}
\text{Exp} & ::= \text{Exp} \text{ AddOp } \text{Exp} \\
\text{Exp} & ::= \text{Term} \\
\text{Term} & ::= \text{Term} \text{ MulOp } \text{Term} \\
\text{Term} & ::= \text{Factor} \\
\text{Factor} & ::= '(\text{Exp}\text{Exp})' \\
\text{Factor} & ::= \text{num} \mid \text{id} \\
\text{AddOp} & ::= '+' \mid '-' \\
\text{MulOp} & ::= '*' \mid '/'
\end{align*}
\]
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 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 forms for if:
  – if \( x < 0 \) then \( x = -x \)
  – if \( x < 0 \) then \( x = -x \) else \( x = x+1 \)

• There is a standard rule for determining which if expression an else clause attaches to
  – If \( x < 0 \) then if \( y < 0 \) \( x = -1 \) else \( x = -2 \)

• The rule:
  – An else clause attaches to the nearest if to the left that does not yet have an else clause
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
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>
Syntax Graphs

Syntax Graphs - Put the terminals in circles or ellipses and put the nonterminals in rectangles; connect with lines with arrowheads

e.g., Pascal type declarations

Provides an intuitive, graphical notation.
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.
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

• How hard is the parsing task?
• Parsing an arbitrary context free grammar is $O(n^3)$ in the worst case.
• E.g., it can take time proportional to the cube of the number of symbols in the input.
• So what?
• This is bad!
Parsing complexity

• If it takes $t_1$ seconds to parse your C program with $n$ lines of code, how long will it take if you make it twice as long?
  
  – $\text{time}(n) = t_1$, $\text{time}(2n) = 2^3 \times \text{time}(n)$
  – 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
Linear complexity parsing

• Practical parsers have time complexity that is linear in the number of tokens, i.e., O(n)
• If v2.0 of your program is twice as long, it will take twice as long to parse
• This is achieved by modifying the grammar so it can be parsed more easily
• 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.
Recursive Decent Parsing

• Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all sentential forms that the nonterminal can generate.

• The recursive decent parsing subprograms are built directly from the grammar rules.

• Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars (why not?)
Hierarchy of Linear Parsers

- Basic containment relationship
  - All CFGs can be recognized by LR parser
  - Only a subset of all the CFGs can be recognized by LL parsers
Recursive Decent Parsing Example

Example: For the grammar:

\[ \text{<term> } \rightarrow \text{<factor> } \{ (\ast | / ) \text{<factor>} \} \]

We could use the following recursive descent parsing subprogram

```c
void term() {
    factor();     /* parse first factor*/
    while (next_token == ast_code ||
        next_token == slash_code) {
        lexical();  /* get next token */
        factor();   /* parse next factor */
    }
}
```
The Chomsky hierarchy

- The Chomsky hierarchy has four types of languages and their associated grammars and machines.
- They form a strict hierarchy; that is, regular languages < context-free languages < context-sensitive languages < recursively enumerable languages.
- The syntax of computer languages are usually describable by regular or context-free languages.
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.