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 more formal techniques for specifying the syntax and semantics

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

Formal Grammar

- A (formal) grammar is a set of rules for strings in a formal language
- The rules describe how to form strings from the language’s alphabet that are valid according to the language’s syntax
- A grammar does not describe the meaning of the strings or what can be done with them in whatever context — only their form

Grammars

Context-Free Grammars
- Developed by Noam Chomsky in the mid-1950s.
- 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

BNF (continued)

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.

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

This is a rule; it describes the structure of a while statement

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.
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.

A non-terminal can have more than one rule

Multiple rules can be combined with the vertical-bar ( | ) symbol (read as “or”)

These two rules:

\<value> ::= \<const>
\<value> ::= \<ident>

are equivalent to this one:

\<value> ::= \<const> | \<ident>

Non-terminals, pre-terminals & terminals

A non-terminal symbol is any symbol that is in 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 a 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 one that appears as a LHS of rule(s), but in every case, the RHSs consist of single terminal symbol, e.g., \<digit> in

\<digit> ::= 0 | 1 | 2 | 3 ... 7 | 8 | 9

BNF

Repetition is done with recursion

E.g., Syntactic lists are described in BNF using 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 subset of English

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

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

Derivations

A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence consisting of just 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

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

\<sentence> -> \<nounPhrase>\<verbPhrase>.
\<article>\<noun>\<verbPhrase>.
the\<noun>\<verbPhrase>.
the man \<verbPhrase>.
the man \<verb>\<nounPhrase>.
the man eats \<nounPhrase>.
the man eats \<noun>.
the man eats the \<noun>.
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
<program> -> <stmts>
<stmts> -> <stmt> ; <stmts>
<stmt> -> <var> = <expr>
<var> -> a | b | c | d
<expr> -> <term> + <term> | <term>
<term> -> <var> | const
Here is a derivation:
<program> => <stmts> => <stmt> => <var> = <expr> => a = <expr> => a = <var> + <term> => a = b + <term> => a = b + const

Finite and Infinite languages
• A simple language may have a finite number of sentences
  – The set of strings representing integers between -10^6 and +10^6 is a finite language
  – 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
• Even very simple grammars may yield infinite languages

Is English a finite or infinite language?
• Assume we have a finite set of words
• Consider adding rules like the following to our previous “baby English” grammar
  <sentence> ::= <sentence><conj><sentence>.
  <conj> ::= and | or | because
• Hint: Whenever you see recursion in a BNF it’s likely that the language is infinite.
  – When might it not be?
  – The recursive rule might not be reachable.

Parse Tree
A parse tree is a hierarchical representation of a derivation

Another Parse Tree
Derivations and Parse Trees

• Note that an unannotated derivation (i.e., just the sequence of sentential forms) contains some, but not all, information about the parse structure
  • Sometimes, it's ambiguous which symbol was replaced
  • Also note that a parse tree contains partial information about the derivation: specifically, what rules were applied, but not the order of application

Grammar

• 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
    • Can you guess why?
  • We can eliminate ambiguity by revising 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 buffalo buffalo Buffalo buffalo

An ambiguous grammar

Here is a simple grammar for expressions that is ambiguous

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

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> -> 1 <op> <e> 
  -> 1 + <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: 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 result 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Pascal</th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>**</code></td>
<td><code>**</code></td>
<td><code>**</code></td>
<td><code>**</code></td>
</tr>
<tr>
<td><code>/</code>, <code>/</code></td>
<td><code>/</code>, <code>/</code></td>
<td><code>/</code>, <code>/</code></td>
<td><code>/</code>, <code>/</code></td>
</tr>
<tr>
<td><code>-</code>, <code>-</code></td>
<td><code>-</code>, <code>-</code></td>
<td><code>-</code>, <code>-</code></td>
<td><code>-</code>, <code>-</code></td>
</tr>
<tr>
<td><code>=</code>, <code>==</code>, <code>!=</code>, <code>!=</code>, <code>==</code>, <code>!=</code></td>
<td><code>=</code>, <code>==</code>, <code>!=</code>, <code>!=</code>, <code>==</code>, <code>!=</code></td>
<td><code>=</code>, <code>==</code>, <code>!=</code>, <code>!=</code>, <code>==</code>, <code>!=</code></td>
<td></td>
</tr>
</tbody>
</table>

Operator Precedence: Precedence Table

<table>
<thead>
<tr>
<th>Operator Precedence: Precedence Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;</code> (bitwise AND)</td>
</tr>
<tr>
<td>`</td>
</tr>
<tr>
<td><code>^</code> (bitwise XOR)</td>
</tr>
<tr>
<td><code>&lt;&lt;</code>, <code>&lt;&lt;</code>, <code>&lt;&lt;&lt;</code>, <code>&lt;&lt;</code>, <code>&lt;&lt;&lt;</code>, <code>&lt;&lt;</code></td>
</tr>
<tr>
<td><code>&gt;</code>, <code>&gt;</code>, <code>&gt;</code>, <code>&gt;</code>, <code>&gt;</code>, <code>&gt;</code></td>
</tr>
<tr>
<td><code>&lt;=</code>, <code>&lt;=</code>, <code>&lt;=</code>, <code>&lt;=</code>, <code>&lt;=</code>, <code>&lt;=</code></td>
</tr>
</tbody>
</table>

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. 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 ** 6.
- Concepts:
  - Explaining rules in terms of operator precedence and associativity
  - Realizing the rules in grammars
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 = 4096
- 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?

Grammar (continued)

Operator associativity can also be indicated by a grammar:

An unambiguous expression grammar:

Sentence: const – const / const

Derivation:
<expr> => <expr> + <term> | const
<expr> => <expr> + const | const
<term> => <term> / const | const
<term> => const

Parse tree:

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.

\[
\begin{align*}
\text{Exp} &::= \text{num} \\
\text{Exp} &::= \text{id} \\
\text{Exp} &::= \text{UnOp Exp} \\
\text{Exp} &::= \text{Exp BinOp Exp} \\
\text{Exp} &::= \text{UnOp} \text{Exp} \\
\text{UnOp} &::= '+' \\
\text{UnOp} &::= '-' \\
\text{BinOp} &::= '+' | '-' | '*' | '/'
\end{align*}
\]

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:

\[
\text{Exp} \Rightarrow \text{Exp BinOp Exp} \Rightarrow \text{id BinOp Exp} \Rightarrow \text{id + Exp} \Rightarrow \text{id + Exp BinOp Exp} \Rightarrow \text{id + id BinOp num} \Rightarrow \text{id + id * num} \Rightarrow a + b * 2
\]

A parse tree

A parse tree for \(a+b*2\):

```
/  \  |
|   |  \
|   |   |
|   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   |   |   |
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 ')' | num | id
AddOp ::= '+' | '-'
MulOp ::= '*' | '/'

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

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

Example: conditionals

The final unambiguous grammar:

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

• The grammar is ambiguous. The second 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

• Goal: write a grammar that forces an else clause to attach to the nearest if w/o an else clause

Grammar

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

Derivation

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

Parse tree

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 if y < 0 x = -1 else x = -2
### 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 affecting what can be done
- All applications of the construct can be systematically replaced with equivalents that don’t use it

*adapted from [Wikipedia](https://en.wikipedia.org)*

### 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 ([ ])
  ```plaintext
  <proc_call> -> ident [ ( <expr_list>)]
  ```
- Put alternative parts of RHSs in parentheses and separate them with vertical bars
  ```plaintext
  <term> -> <term> (+ | -) const
  ```
- Put repetitions (0 or more) in braces ({ } )
  ```plaintext
  <ident> -> letter {letter | digit}
  ```

### 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.

### 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 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(n) 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 to 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 or 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

### 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 descent 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:  

\[
<\text{term}>::=<\text{factor}>\ (\epsilon/\ |/ <\text{factor}>)
\]

We could use the following recursive descent parsing subprogram (e.g., one in C)

```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

<table>
<thead>
<tr>
<th>Machines</th>
<th>Grammars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turing machines</td>
<td>Phrase structure</td>
</tr>
<tr>
<td>Linear-bounded</td>
<td>Context-sensitive</td>
</tr>
<tr>
<td>Push-down</td>
<td>Context-free</td>
</tr>
<tr>
<td>Finite state</td>
<td>Regular</td>
</tr>
</tbody>
</table>

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.