Syntax and Parsing

- **Syntax**: the form of a program

- **Semantics**: the meaning of a program

- **Two parts to syntax analysis**:  
  - lexical rules: define legal characters and how they can be combined to form symbols ("lexemes").
  - syntax rules: define how categories of lexemes ("tokens") can be combined to form legal programs.
Syntax and Parsing

- More formal definition of Syntax:
  - Syntax defines the symbols of a language, and the spatial relationships between symbols in that language

- Semantics
  - Semantics describes the meanings to be ascribed to symbols, and structures of symbols, in a language.
We won't make a strict distinction, but will generally deal with syntax rules.
How to Describe the Syntax of a Language?

- English description
  - lengthy, tedious, ambiguous

- Formal description
  - recognizer: given a string, a recognizer for a language L tells whether or not the string is in L (ex: Compiler)
  - generator: a generator for L will produce an arbitrary string in L on demand. (ex: Grammar)

- Recognition and generation are useful for different things, but are closely related.

- First, we'll talk about an important generation tool: BNF.
BNF - A Generative Tool

- Backus-Naur Form (BNF) is a *metalanguage* for describing the syntax of programming languages.
  - developed by John Backus and Peter Naur
  - first used to describe ALGOL60

- A language description in BNF is called a *grammar*.

Note: In formal linguistics a grammar is described as the syntax of a language together with an knowledge of the allowable constructs that provide “sensible” meanings. Obviously the syntax of a programming language does not contain “knowledge”, but we persist in this misuse of the term “grammar”.

Grammars

- A grammar is made up of productions, or rules, e.g.,
  
  \[
  \text{<sentence>} \rightarrow \text{<subject>}\text{<verb>}\text{<object>}
  \]
  
  \[
  \text{<verb>} \rightarrow \text{see} | \text{love}
  \]
  
  \[
  \text{<subject>} \rightarrow \text{I}
  \]
  
  \[
  \text{<object>} \rightarrow \text{him} | \text{her}
  \]

- Five components:
  
  The rules using:
  
  \[
  \rightarrow : \text{"is defined as"}
  \]
  
  \[
  | : \text{"or"}
  \]

  implied concatenation
  
  terminals: see, love, I, him, her.
  
  non-terminals: <sentence>, <verb>, <subject>, <object>
  
  root symbol: <sentence>
  
  the algorithm of generation or analysis

Note: Except where specifically required, spaces are assumed to be inserted between “lexemes” in natural languages.
Recursion

- Need recursion to define strings of indefinite length:
  
  \[ \text{<ones>} \to 1 | 1\text{<ones>}, \]
  \[ \to 1, 11, 111, 1111, \ldots \]
  
  \[ \text{<ablist>} \to ab | a\text{<ablist>}b, \]
  \[ \to ab \]
  \[ \text{OR } a\text{<ablist>}b, \]
  \[ \to \text{ab} \]
  \[ \text{OR } a\text{<ablist>}b, \]
  \[ \to \text{aabb} \]
  \[ \text{OR } a\text{a<ablist>}bb, \]
  \[ \to \text{aaabbb} \]
  \[ \text{OR } a\text{a<ablist>}bb, \]
  \[ \to \text{aaa<ablist>}bbb, \]
  \[ \ldots \]

Describe the language resulting from this syntax!

Write a syntax that describes a language composed of strings of arbitrary numbers of “a”s followed by arbitrary numbers of “b”s
Derivations

The steps in generating a string using a grammar are called a derivation.

\[
\begin{align*}
& <\text{exp}> \rightarrow <\text{id}> \mid <\text{exp}> + <\text{exp}> \mid <\text{exp}> * <\text{exp}> \mid (<\text{exp}>)
\end{align*}
\]

\[
\begin{align*}
& <\text{id}> \rightarrow A \mid B \mid C
\end{align*}
\]

Each step produces a sentential form. It is a string of terminals and nonterminals.

A sentence is a sentential form containing only terminals.
Language

- We are now ready to define a LANGUAGE:
  - A language is a set of strings of terminals.

- **Note**
  - A language may be finite or infinite. The empty set is a language.

- **So now define the LANGUAGE GENERATED BY A GRAMMAR:**
  - The language generated by a grammar is the set of sentences derivable using the grammar.
Left-Most and Right-Most Derivations

- How to choose which non-terminal to replace next:
  - left-most derivation: replace left-most NT first
  - right-most derivation: replace right-most NT first

- Need not be either of these
  - random replacement OK
    can't affect language generated
Parse Trees

- Show the **syntactic** structure of sentences.
Ambiguous Grammars

- A grammar is ambiguous if it generates the same sentence for which there are two or more parse trees.

- Another parse tree for $A \times B + C$:
Disambiguating the Grammar

- To disambiguate this grammar, change to:
  \[
  \begin{align*}
  \text{<exp>} & \rightarrow \text{<exp>} + \text{<term>} | \text{<term>} \\
  \text{<term>} & \rightarrow \text{<term>} * \text{<factor>} | \text{<factor>} \\
  \text{<factor>} & \rightarrow (\text{<exp>}) | \text{<id>} \\
  \text{<id>} & \rightarrow \text{A} | \text{B} | \text{C}
  \end{align*}
  \]

- This forces * to have a higher precedence than +, even though it generates the same language as the first grammar.
Chomsky Hierarchy of Syntax

- **0: Unrestricted grammar rules**
  \[ \text{string} \rightarrow \text{string} \]

- **1: Context Sensitive**
  \[ \text{ABC} \rightarrow \text{A} \gamma \text{C} \]

- **2: Context Free**
  \[ \text{A} \rightarrow \text{string} \]

- **3: Linear**
  \[ \text{A} \rightarrow \text{aB} | \text{b} \]

where

- \( N \) - set of non-terminals
- \( T \) - set of terminals
- \( A, B, C, \gamma \in N \)
- \( a, b \in T \)
- string \( \epsilon \) \((N \cup T)^* \)
Chomsky Hierarchy of Syntax

- **0: Unrestricted grammar**
  - string → string (arbitrary word processing change)

- **1: Context Sensitive**
  - ABC → AγC (change B to γ in the context of A..C)

- **2: Context Free → syntactic analyzer**
  - A → string (change A to string anywhere)

- **3: Linear → lexical analyzer**
  - A → aB | b (there is a one-for-one replacement in N)
Chomsky Hierarchy of Syntax

\[ G_0 \supseteq G_1 \supseteq G_2 \supseteq G_3 \]

- \( G_3 \) - linear, one-dimensional
- \( G_2 \) - two-dimensional
- \( G_1 \) - three-dimensional
Another Ambiguous Grammar

<stmt>  -->  <assign> | <if_stmt>
<assign> --> <id> := <exp>
<if_stmt>  -->  IF <bool> THEN <stmt> | IF <bool> THEN <stmt> ELSE <stmt>

- Exercise:
  - Prove that this is ambiguous.
  - Write a grammar for the same language that is not ambiguous. (Use any published reference material)

Note: This grammar (syntax) develops a language that demonstrates the “DANGLING ELSE” problem that occurred in ALGOL 60.
Limitations of Context Free Grammars

- Productions must always apply, regardless of context in which string appears.

- Can't handle some things:

  var x : integer;
  y : boolean;
  begin
    x := 3; OK
    x := y; types wrong
    z := 5; var z undeclared

- Need "static semantics" . . .
Recognizers

- How to generate a recognizer from a grammar?
  - automatically (YACC)
  - by hand

- There are many types of parsers:
  - LL(0)
  - LL(k)
  - LR
  - LALR
  - recursive descent
Extended BNF & Regular Expressions

Bracket notations

- \([. . .]\)  optional
  \(<\text{if}>\) --> IF \(<\text{bool}>\) THEN \(<\text{stmt}>\) [ELSE \(<\text{stmt}>\)]

- \({. . .}\)  zero or more times
  \(<\text{ones}>\) --> 1 \<{\text{ones}>\}

- \(. . . | . . .\) or \((:::)\)  local choice
  \(<\text{exp}>\) --> \(<\text{id}>\) | \(<\text{exp}>\) (+ | *) \(<\text{exp}>\)

Regular Expressions

- \(a^*\) equivalent to \{a\} - zero or more times
- \(a^+\) equivalent to \(a\{a\}\) - one or more times
Syntax and Parsing Summary

- Recognizer vs. generator
- BNF
  - Four components
  - Recursion
- Derivation
- Ambiguous grammars
- Extended BNF
Semantics of Programming Languages

- How to define the meaning of programs?

- Three approaches:
  - Operational
  - Axiomatic
  - Denotational
Operational Semantics

- Gives a program’s meaning in terms of its implementation on a real or virtual machine

- Define two parts:
  - machine
    - high level
    - low level
  - translation from source code to "machine" code

- Let S be the state, having components mem, input, and output --- where mem consists of register values, memory values, etc.

  $S \{ \text{instruction} \} S'$

  $\text{Diff} \ (S, S') \Rightarrow \text{“meaning of instruction”}$
### Example

<table>
<thead>
<tr>
<th>Pascal</th>
<th>Operational Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for i := x to y do</code></td>
<td><code>i := x</code></td>
</tr>
<tr>
<td><code>begin</code></td>
<td><code>loop: if i&gt;y goto out</code></td>
</tr>
<tr>
<td>`.</td>
<td>.</td>
</tr>
<tr>
<td>`.</td>
<td>.</td>
</tr>
<tr>
<td>`.</td>
<td>.</td>
</tr>
<tr>
<td><code>end</code></td>
<td><code>i := i + 1</code></td>
</tr>
<tr>
<td></td>
<td><code>goto loop</code></td>
</tr>
<tr>
<td></td>
<td><code>out: . . .</code></td>
</tr>
</tbody>
</table>

- Operational semantics could be much lower level, e.g.,

```plaintext
mov i,r1
mov y,r2
jmpifless(r2,r1,out)
...
out: ...
```
Advantages and Disadvantages of Operational Semantics

- **Advantages:**
  - May be simple, intuitive for small examples
  - Useful for implementation

- **Disadvantages**
  - Very complex for large programs
  - Lacks mathematical rigor

- **Uses:**
  - Vienna Definition Language (VDL) used to define PL/I (Wegner 1972)
  - Compiler work
Axiomatic Semantics

- Based on predicate calculus. Use assertions to show certain properties of programs.

\[ \{P\} \text{ statement } \{Q\} \]

- Compute precondition from postcondition:

\[ \{P\} \ x := y + 1 \ \{x>0\} \]

→ Possible Ps:

\[ y > 5 \]
\[ y = 37 \]
\[ y \geq 0 \]

→ Weakest Precondition (WP)

etc.

- **WP**-\(\rightarrow\) identifies *all possible* cases for which postcondition holds!

\[ \text{WP } \Rightarrow \text{ Least Restrictive} \]
Finding the Weakest Precondition

- Define function:

  \[ \text{wp: Stmt} \times \text{Postcondition} \rightarrow \text{weakest precondition} \]

  \[ \text{wp} \left( x := e, P \right) = P_{x \rightarrow e} \quad \text{"substitute } e \text{ for every } x \text{ in } P \" \]

- So:

  \[ \text{wp} \left( x := y+1, x > 0 \right) \]
  \[ = x > 0_{x \rightarrow y+1} \]
  \[ = y+1 > 0 \]
  \[ = y \geq 0 \]

  \[ \rightarrow \quad \text{basically, "undoing" the assignment and solving for } y \]

  \[ y > -1 \text{ is WP, too} \]

  \[ y > -1 \land y \geq 0 \text{ when } y \text{ is an integer} \]
Sequences of Statements

\{P\} S1; S2 \{Q\}

- Just apply \(\text{wp}\) twice

\[
\begin{align*}
\text{wp} (x := y + 1; z := x + y, z > 5) \\
\text{wp} (z := x + y, P1) \\
&= z > 5_{z \rightarrow x + y} \\
&= x + y > 5
\end{align*}
\]

What can we say about \(x+y\) knowing that after the assignment is made \(z > 5\)?

\[
\begin{align*}
\text{wp} (x := y + 1, x + y > 5) \\
&= x + y > 5_{x \rightarrow y + 1} \\
&= y + 1 + y > 5 \\
&= y > 2
\end{align*}
\]
Loops

- \{P\} while B do S end \{Q\}

- Need *loop invariant* \(I\) such that:
  
  \[
  \begin{align*}
  &\implies P \implies I \\
  &\implies \{I\} B \{I\} \\
  &\implies \{I \land B\} S \{I\} \\
  &\implies (I \land \text{(not } B)) \implies Q \\
  &\implies \text{and the loop terminates}
  \end{align*}
  \]
Finding Loop Invariants

- Work backwards through a few iterations and look for a pattern.

```plaintext
while y <> x do y := y+1 {y = x}
```

\[
wp (y := y + 1, \{y = x\}) = \{y = x\}_{y \rightarrow y+1}
\]

\[
= y = x - 1
\]  
-- last time through

\[
wp (y := y + 1, \{y = x - 1\}) = \{y = x-1\}_{y \rightarrow y+1}
\]

\[
= y = x - 2
\]  
-- next to last time

Now, by extension we get  \( I = \{ y < x \} \)

When we factor in that the loop may not be executed even once (implying \( wp \) could be \( y = x \)) then we get

\[ I = \{ y \leq x \} \]

- This also satisfies loop termination, so

\[ P = I = \{y \leq x\} \]
Finding Loop Invariants (cont.)

- Does \( \{y \leq x\} \) satisfy conditions (1) - (5)?

  1. \( y = x \implies y \leq x \) ?

  2. if \( y \leq x \) and \( y \neq x \) is then evaluated, is it still the case that \( y \leq x \) ?

  3. if \( y \leq x \) and \( y \neq x \) are both true and then \( y := y+1 \) is executed, then is \( y \leq x \) ?

  4. does \( y \leq x \) and \( y = x \implies y = x \) ?

  5. Can you argue convincingly that the program segment terminates?
Finding Loop Invariants (a harder example)

- However, finding the WP is not always so easy! Consider:

\[
\{P\} \text{ while } y < x + 1 \text{ do } y := y + 1 \{y>5\}
\]

\[
y > 5\quad y \rightarrow y + 1 \quad \Rightarrow \quad y > 4
\]
\[
y > 4\quad y \rightarrow y + 1 \quad \Rightarrow \quad y > 3
\]

etc.

- really tells us *nothing* relative to \( x \) because \( x \) is not in

\[
Q \equiv \{y > 5\}
\]

- Try Using Boolean

\[
I \land (\neg B) \Rightarrow Q
\]
\[
? \land y \geq x + 1 \Rightarrow y > 5
\]
\[
? \land y > x \quad \Rightarrow \quad y > 5
\]

any \( x \geq 5 \) satisfies implication

so . . . let \( I \equiv x \geq 5 \)

- Do the 4 Axioms hold?
Advantages, Disadvantages, and Uses of Axiomatic Semantics

- **Advantages**
  - Can be very abstract
  - May be useful in proofs of correctness
  - Solid theoretical foundations

- **Disadvantages**
  - Predicate transformers are hard to define
  - Hard to give complete meaning
  - Does not suggest implementation

- **Uses of Axiomatic Semantics**
  - Semantics of Pascal
  - Reasoning about correctness
HOMEWORK FOR AXIOMATIC SEMANTICS

Consider

\{P\} \quad \text{\(X := X \times 3\)} \quad \text{\(X^2 = 36\)}

Determine \textbf{Weakest Precondition} \(P\)
Denotational Semantics

- Define a function that maps a program (a syntactic object) to its meaning (a semantic object).

- Sort of like a high-level operational semantics, except
  - machine is gone
  - language is $\lambda$-calculus

- More abstract.
Example: Decimal Numbers

Valuation function: \( V: \text{Number} \rightarrow \text{Integers} \)

- **Syntax:**
  \[
  \langle \text{num} \rangle \rightarrow \langle \text{num} \rangle \langle \text{digit} \rangle | \langle \text{digit} \rangle \\
  \langle \text{digit} \rangle \rightarrow 0 | 1 | 2 | \ldots | 9
  \]

- **Semantics:**
  \[
  \text{Let } n \in \langle \text{num} \rangle, \ d \in \langle \text{digit} \rangle \\
  V[n]_{nd} = 10 \cdot V[n]_n + V[d]_d \\
  V[0] = 0 \\
  V[1] = 1
  \]

- **Consider** \( V[237] \):
  \[
  = 10 \cdot (10 \cdot 2 + 3) + 7 \\
  = 10 \cdot (20 + 3) + 7 \\
  = 10 \cdot (23) + 7 \\
  = 230 + 7 \\
  = 237
  \]

Integers as we know them.
Expressions

- But for real programming languages we need more info:
  \[ E: \text{Expression} \rightarrow \text{Integer} \]

  \[ E(\llbracket x \rrbracket) = ?\quad \text{where x is a variable} \]

- Depends on the current state

  \[ \text{STATE} = <\text{mem}, \text{input}, \text{output}> \]

  mem: Identifier \rightarrow Integer

  input: Integer *

  output: Integer *

  Streams of integers

- Now

  \[ E: \text{Expression} \times \text{STATE} \rightarrow \text{Integer} \]

  \[ E(\llbracket x \rrbracket, s) = \text{mem}(\llbracket x \rrbracket) \quad \text{where } s = <\text{mem},i,o> \]

  \[ E(\llbracket e_1 + e_2 \rrbracket, s) = E(\llbracket e_1 \rrbracket, s) + E(\llbracket e_2 \rrbracket, s) \]
Expressions denote a value, but statements denote a state.

\[ \text{ST: } (\text{Stmt } x \text{ STATE}) \rightarrow \text{STATE} \]

\[ \text{ST } (\begin{array}{c} x := e \end{array}, s) = \langle \text{mem}',i,o \rangle \text{ where} \]

\[ s = \langle \text{mem},i,o \rangle \]

\[ \text{mem}'[x] = E[\begin{array}{c} e \end{array},s] \]

\[ \text{mem}'[y] = \text{mem}[y] \quad \text{for all } y \neq x \]

\[ \text{ST } (\begin{array}{c} \text{write}(e) \end{array}, s) = \langle \text{mem},i,o' \rangle \text{ where} \]

\[ s = \langle \text{mem},i,o \rangle \]

\[ o' = o \circ (E[\begin{array}{c} e \end{array}, s]) \]
Sequences of Statements

- **Basic (sequential statement evaluation)**

  \[
  \text{ST}(\llbracket \text{stmt}_1; \text{stmt}_2 \rrbracket, s) = \text{ST}(\llbracket \text{stmt}_2 \rrbracket, s') \text{ where } \\
  s' = \text{ST}(\llbracket \text{stmt}_1 \rrbracket, s)
  \]

- **Parallel statement evaluation**

  \[
  \text{ST}(\llbracket \text{stmt}_1; \text{stmt}_2; \text{stmt}_3 \rrbracket, s) = \{s_1, s_2, s_3\} \text{ where } \\
  s_1 = \text{ST}(\llbracket \text{stmt}_1; \text{stmt}_2; \text{stmt}_3 \rrbracket, s) \\
  s_2 = \text{ST}(\llbracket \text{stmt}_1; \text{stmt}_3; \text{stmt}_2 \rrbracket, s) \\
  s_3 = \text{ST}(\llbracket \text{stmt}_3; \text{stmt}_1; \text{stmt}_2 \rrbracket, s)
  \]
Example

\[
P: \{ x := 5; \}
\]

\[
P': \{ y := x + 1; \} \quad \text{write}(x \ast y); \} \quad P''
\]

\[\rightarrow \quad \text{Initial state } s = <\text{mem}, i, o>\]

\[
\begin{align*}
\text{ST}(\llbracket P \rrbracket, s) &= \text{ST}(\llbracket P' \rrbracket, (\text{ST}(\llbracket x := 5 \rrbracket, s))) \\
s' &= <\text{mem}', i', o'> \quad \text{where} \\
\text{mem}'(\llbracket x \rrbracket) &= 5 \\
\text{mem}'(\llbracket z \rrbracket) &= \text{mem}(\llbracket z \rrbracket) \quad \text{for all } z \neq x \\
i' &= i, o' = o
\end{align*}
\]
Example (continued)

\[ \rightarrow \text{ST}(\llbracket P' \rrbracket, s') = \text{ST}(\llbracket P'' \rrbracket, (\text{ST}(\llbracket y := x + 1 \rrbracket, s'))) \]

\[ s'' = <\text{mem}'', i'', o''> \text{ where} \]
\[ \text{mem}''(\llbracket y \rrbracket) = E(\llbracket x + 1 \rrbracket, s') = 6 \]
\[ \text{mem}''(\llbracket z \rrbracket) = \text{mem}'(\llbracket z \rrbracket) \text{ for all } z \neq y \]
\[ i'' = i', o'' = o' \]

\[ \rightarrow \text{ST}(\llbracket P'' \rrbracket, s'') = \text{ST}(\llbracket \text{write} \ (x \ast y) \rrbracket, s'') = s''' \]

\[ s''' = <\text{mem}''', i''', o'''> \text{ where} \]
\[ \text{mem}''' = \text{mem}'', i''' = i'' \]
\[ o''' = o'' \ast E(\llbracket x \ast y \rrbracket, s'') = o'' \ast 30 \]

\[ \rightarrow \text{So,} \]
\[ \text{ST}(\llbracket P \rrbracket, s) = <\text{mem}''', i''', o''' > \text{ where} \]
\[ \text{mem}'''(\llbracket y \rrbracket) = 6 \]
\[ \text{mem}'''(\llbracket x \rrbracket) = 5 \]
\[ \text{mem}'''(\llbracket z \rrbracket) = \text{mem}(\llbracket z \rrbracket) \text{ for all } z \neq x, y \]
\[ i''' = i \]
\[ o''' = o \ast 30 \]
Advantages, Disadvantages, and Uses of Denotational Semantics

- Advantages (of denotational semantics)
  - compact and precise
  - may help with implementation
  - solid mathematical foundations

- Disadvantages
  - Hard for programmer to use

- Uses
  - Semantics for Algol-60, Pascal, etc.
  - Compiler generation and optimization
Prefatory Consideration:

Prog. Langs. have conditional statements, e.g.

1) if b then stmt1, else stmt2

2) if b then exp1, else exp2

Assuming that conditionals only support expression evaluation and have no side effects, let’s give meaning to 1) above:

\[
\text{ST}(\text{if } b \text{ then } \text{stmt}, \text{ else } \text{stmt2}, s) = \\
\text{if } E(b, s) \text{ then} \\
\text{ST}(\text{stmt1}, s) \text{ else} \\
\text{ST}(\text{stmt2}, s).
\]

Note: use of previous defns T/F Assessment like in ‘C’ introduction of”"IF THEN ELSE” in denotational language
UNDERSTAND/STUDY

1) ST (if b then stmt1 else stmt2, s)
   definition and elaboration

2) Give denotational semantics for repeat until stmt

   REPEAT stmt UNTIL b

   You will need conditional statement like that specified above

HINT:

   on RHS you might find recursive defn
Summary

● Each form of semantic description has its place:
  → **Operational**
    - informal descriptions
    - compiler work
  → **Axiomatic**
    - reasoning about particular properties
    - proofs of correctness
  → **Denotational**
    - formal definitions
    - provably correct implementations