Syntax

The syntax of a language (formal or natural) is the way the words in a sentence/program can be arranged.

- eats dog bone the is not a legal arrangement of words in English.
- \( y x + 5 \) is not a legal arrangement of tokens in Java.
- Somehow, we need to describe what constitutes legal and illegal sentences in a particular language.
- We use production rules to describe the syntax of a language.

Production Rules

Here's a production rule:

\[
\text{IfStat} \rightarrow \text{if} \ ( \text{expr} \ ) \ \text{stat}
\]

This rule states that to construct an if-statement in C you have to type

1. an if, then
2. a (, then
3. some sort of expression, then
4. a ), then finally
5. some sort of statement.

A grammar can be used for

- sentence generation (i.e. which sentences does this grammar generate?), or
- parsing (i.e. is sentence \( S \) generated by this grammar?).

Let's look at a simple grammar for a fragment of English.
Syntactic Categories

A Simple English Grammar

S  [Sentence]  John likes Sarah’s black hair
N  [Noun]  John, hair
V  [Verb]  eating, sat
Adj  [Adjective]  black, long
Det  [Determiner]  the, a, every
NP  [Noun Phrase]  Sarah’s long black hair
VP  [Verb Phrase]  eating apples

S  →  NP VP
NP  →  N
N  →  John
N  →  Lisa
N  →  house
VP  →  V NP
V  →  died
V  →  kissed
NP  →  N
NP  →  Det N
Det  →  the
Det  →  a

S, NP, VP, N, Det, V are non-terminal symbols.
John, Lisa, house, died, ... are terminal symbols.
S is the start symbol.

Sentence Generation

 Terminology

A grammar is a 4-tuple
(non-terminals, terminals, productions, start-symbol)
or
(N, Σ, P, S)
A production is of the form  α → β where  α, β  are taken from  N ∪ Σ.
Read  α → β  as “rewrite  α  with  β”.
Read  ⇒  as “directly derives”.
Read  ⇒ r as “directly derives using rule  r”.
Read  ⇒ * as “derives in one or more steps”.

1. Start with the start symbol.
2. Pick a non-terminal  X  on the right hand side.
3. Pick a grammar rule  X → γ.
4. Replace  X  with  γ.
5. Repeat until left with a string of words.
Here's a grammar for a simple programming language:

```
Program ::= BEGIN Stat END
Stat ::= ident := Expr
Expr ::= Expr + Expr | Expr * Expr | ident | number
```

- We write terminal symbols like `this`.
- We write non-terminal symbols like `this`.
- Sometimes we write `::=` instead of `→`.
- \( A → b | c \) is the same as \( A → b ; A → c \). Read `|` as "or".

We know the sentence

```
BEGIN a := 5 + 4 * 3 END
```

is in the language because we can derive it from the start symbol:

```
Program ⇒ BEGIN Stat END
⇒ BEGIN ident := Expr END
⇒ BEGIN "a" := Expr END
⇒ BEGIN "a" := 5 + Expr END
⇒ BEGIN "a" := 5 + Expr * Expr END
⇒ BEGIN "a" := 5 + 4 * Expr END
⇒ BEGIN "a" := 5 + 4 * 3 END
```

**Terminology...**

- Our English grammar is the 4-tuple
  \( (\{S,NP,V,\ldots\},\{John,house,died,\ldots\},\{S → NP VP, VP → V,\ldots\},S) \)

- Our PL grammar is the 4-tuple
  \( (\{Program,Stat,\ldots\},\{BEGIN,=:,*,\ldots\},\{Program ::= BEGIN Stat END,\ldots\},Program) \)

**Parse Trees**

- We often want to show how a particular sentence was derived. We can do this without listing all the steps explicitly by drawing a parse tree.
- A parse tree is a tree where
  1. The root is labeled by the start symbol.
  2. Each leaf is labeled by a terminal symbol.
  3. Each interior node is labeled by a non-terminal symbol.
If one step of our derivation is

\[ \cdots A \cdots \Rightarrow \cdots X Y Z \cdots \]

(i.e., we used the rule \( A \rightarrow XYZ \)) then we'll get a parse (sub-)tree.

\[ \begin{array}{c}
A \\
\downarrow \\
\cdots \\
\cdots \\
\cdots \\
\cdots \\
\end{array} \]

**Regular Grammars and Lexical Analysis**

Program \( \Rightarrow \) BEGIN Stat END  
\( \Rightarrow \) BEGIN ident := Expr END  
\( \Rightarrow \) BEGIN "a" := Expr END  
\( \Rightarrow \) BEGIN "a" := Expr + Expr END  
\( \Rightarrow \) BEGIN "a" := 5 + Expr END  
\( \Rightarrow \) BEGIN "a" := 5 + 4 * Expr END  
\( \Rightarrow \) BEGIN "a" := 5 + 4 * 3 END  
\( \Rightarrow \) BEGIN ident := Expr END  
\( \Rightarrow \) BEGIN "a" := Expr END  
\( \Rightarrow \) BEGIN "a" := Expr + Expr END  
\( \Rightarrow \) BEGIN "a" := 5 + Expr END  
\( \Rightarrow \) BEGIN "a" := 5 + 4 * Expr END  
\( \Rightarrow \) BEGIN "a" := 5 + 4 * 3 END  

Program  
\( \Rightarrow \) BEGIN Stat END  
\( \Rightarrow \) ident := Expr  
\( \Rightarrow \) "a" := Expr + Expr  
\( \Rightarrow \) "a" := 5 + Expr  
\( \Rightarrow \) "a" := 5 + 4 * Expr  
\( \Rightarrow \) "a" := 5 + 4 * 3  

\[ S \rightarrow NP \quad VP \]
\[ NP \rightarrow N \]
\[ N \rightarrow John \]
\[ VP \rightarrow V \quad NP \]
\[ V \rightarrow \text{kissed} \]
\[ NP \rightarrow N \]
\[ N \rightarrow Lisa \]
Regular Grammars

A grammar is regular if all rules are of the form

\[ A \rightarrow aB \]
\[ A \rightarrow a \]

By convention, the symbols \( A, B, C, \ldots \) are non-terminals, \( a, b, c, \ldots \) are terminals, and \( \alpha, \beta, \gamma, \ldots \) are strings of symbols.

Regular grammars are used to describe the lexical structure of programs, i.e. what tokens look like.

Parsing and the Definition of Syntax

Context-Free Grammars

Programming language syntax is described by a context free grammar (CFG).

In a CFG all rules are of the form

\[ A \rightarrow \gamma \]

\( \gamma \) is any sequence of terminals or non-terminals. \( A \) is a single non-terminal.

Example: an if-statement consists of an if-token, expression, then-token, statement, and (maybe) an else-token followed by a statement.

EBNF

BNF is Backus-Naur Form, a way to write CFGs. EBNF (Extended BNF) is a more expressive way to write CFGs.

Repetition and choice are common structures in a language (and hence, its grammar).

Repetition:

```java
int x,y,z,w,....;
```

Choice:

```java
class C { ... }  
class C extends D { ... }
```
EBNF for Luca

```
program ::= 
    PROGRAM ident ; decl_list block ;

decl_list ::= 
    { declaration ; }

declaration ::= 
    VAR ident : ident | 
    TYPE ident = RECORD [ field_list ] | 
    TYPE ident = ARRAY expression OF ident | 
    CONST ident : ident = expression | 
    PROCEDURE ident ( [formal_list] ) decl_list block ;
```

EBNF for Luca...

```
field_list ::= field_decl { ; field_decl }

field_decl ::= ident : ident

formal_list ::= formal_param { ; formal_param }

formal_param ::= [VAR] ident : ident

actual_list ::= expression { , expression }

block ::= BEGIN stat_seq END

stat_seq ::= { statement ; }
```
EBNF for Luca...

statement ::= 
  designator := expression | 
  WRITE expression | READ designator | WRITELN 
  ident( [ actual_list ])
  IF expression THEN stat_seq [ELSE stat_seq] ENDF | 
  FOR ident := expression TO expression [BY expression] DO 
  stat_seq ENDF | 
  WHILE expression DO stat_seq ENDDO | 
  REPEAT stat_seq UNTIL expression | 
  LOOP stat_seq ENDLOOP | EXIT

expression ::= 
  expression bin_operator expression | unary_operator expression |
  ( expression ) |
  real_literal | integer_literal | char_literal | string_literal | 
  designator |

unary_operator ::= − | TRUNC | FLOAT | NOT

bin_operator ::= + |− |∗ | / | % | < | <= | = | # | >= | > | AND | OR

Ambiguous Grammars

A grammar is ambiguous if some string of tokens can produce two (or more) different parse trees.

PRECEDE & ASSOCIATIVITY & AMBIGUITY

E ::= E + E | E * E | number

\[ 5 + 4 \times 3 \]

\[ E \Rightarrow E + E \]
\[ E \Rightarrow E + E \]
\[ E \Rightarrow E \times E \]
\[ E \Rightarrow E \times E \]
\[ 5 + 4 \times 3 \]
\[ 5 + E \]
\[ 5 + E \]
\[ 5 + 4 \times E \]
\[ 5 + 4 \times 3 \]
\[ 5 + 4 \times 3 \]
\[ 5 + 4 \times 3 \]
\[ 5 + 4 \times 3 \]
\[ 5 + 4 \times 3 \]
\[ 5 + 4 \times 3 \]
Structural Ambiguity in English

- Ambiguities occur in natural languages also:
  - *S* → *NP VP NP*
  - *S* → *NP VP PP NP*

Operator Precedence

- The *precedence* of an operator is a measure of its *binding power*, i.e. how strongly it attracts its operands.
- Usually * has higher precedence than +:
  - $4 + 5 \times 3$
  - means
  - $4 + (5 \times 3)$,
  - not
  - $(4 + 5) \times 3$.
- We say that * binds harder than +.

Operator Associativity

- The *associativity* of an operator describes how operators of equal precedence are grouped.
- + and − are usually *left associative*:
  - $4 - 2 + 3$
  - means
  - $(4 - 2) + 3 = 5$,
  - not
  - $4 - (2 + 3) = -1$.
- We say that + associates to the left.
- ^ associates to the right:
  - $2^3^4 = 2^{(3^4)}$.

Operators in C

<table>
<thead>
<tr>
<th>Operator</th>
<th>Kind</th>
<th>Prec</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[k]</td>
<td>Primary</td>
<td>16</td>
<td>Left</td>
</tr>
<tr>
<td>f(...)</td>
<td>Primary</td>
<td>16</td>
<td>Left</td>
</tr>
<tr>
<td>.</td>
<td>Primary</td>
<td>16</td>
<td>Left</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Primary</td>
<td>16</td>
<td>Left</td>
</tr>
<tr>
<td>a++, a--</td>
<td>Postfix</td>
<td>15</td>
<td>Left</td>
</tr>
<tr>
<td>++a, --a</td>
<td>Unary</td>
<td>14</td>
<td>Left</td>
</tr>
<tr>
<td>-</td>
<td>Unary</td>
<td>14</td>
<td>Left</td>
</tr>
<tr>
<td>!</td>
<td>Unary</td>
<td>14</td>
<td>Left</td>
</tr>
<tr>
<td>&amp;</td>
<td>Unary</td>
<td>14</td>
<td>Left</td>
</tr>
<tr>
<td>*</td>
<td>Unary</td>
<td>14</td>
<td>Left</td>
</tr>
<tr>
<td>/, %, &lt;&lt;, &gt;&gt;</td>
<td>Binary</td>
<td>13</td>
<td>Left</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>Binary</td>
<td>12</td>
<td>Left</td>
</tr>
<tr>
<td>==, !=</td>
<td>Binary</td>
<td>11</td>
<td>Left</td>
</tr>
<tr>
<td>&amp;&amp;,</td>
<td></td>
<td></td>
<td>Binary</td>
</tr>
<tr>
<td>?, :</td>
<td>Binary</td>
<td>9</td>
<td>Left</td>
</tr>
<tr>
<td>=, +=, -=, *=, /=, %=, &lt;&lt;=, &gt;&gt;&gt;=, &amp;=, ^=,</td>
<td>=</td>
<td>Binary</td>
<td>8</td>
</tr>
</tbody>
</table>

- Binary 7: Left
- Binary 6: Left
- Binary 5: Left
- Binary 4: Left
- Ternary 3: Right
- Binary 2: Right
- Binary 1: Left
Abstract Syntax

- We distinguish between a language’s **concrete** and **abstract** syntax.
- The concrete syntax describes the textual layout of programs written in the language, e.g. what *if*-statements look like.
- The abstract syntax describes the **logical** structure of the language; e.g. that *if*-statements consist of three parts (expression, statement, statement).

The abstract syntax also describes the structure of the **abstract syntax tree** (AST).
- Each abstract syntax rule represents the structure of an AST node-type.
- A parser converts from the program’s concrete syntax to its corresponding abstract syntax, i.e. it reads the source code of the input program and produces an AST.

Examples
Concrete Grammar Example I

Grammar Example I

---

Concrete Grammar: 

\[
S ::= \text{ident} := E | \text{if } E \text{ then } SS_1[\text{else } SS_2] \text{ end } | \text{while } E \text{ do } SS \text{ end } | \epsilon \\
SS ::= S ; SS | \epsilon \\
\]

Abstract Grammar: 

\[
\begin{align*}
\text{Assign} & ::= \text{ident} \text{ Expr} \\
\text{If} & ::= \text{Expr} \text{ StatSeq} \\
\text{IfElse} & ::= \text{Expr} \text{ StatSeq} \text{ StatSeq} \\
\text{While} & ::= \text{Expr} \text{ StatSeq} \\
\text{Stat} & ::= \text{Assign} | \text{If} | \text{IfElse} | \text{While} \\
\text{StatSeq} & ::= \text{Stat} \text{ StatSeq} | \text{NULL} \\
\end{align*}
\]

---

Grammar Example I...

- The rule 
  \[
  \text{IfElse} ::= \text{Expr} \text{ StatSeq} \text{ StatSeq} \\
  \]
  says that an if-statement consists of three parts, or, equivalently, that an AST if-node will have three children:

- We use recursive rules to define lists (e.g. declaration-lists, statement-lists):
  \[
  \text{StatSeq} ::= \text{Stat} \text{ StatSeq} | \text{NULL} \\
  \]

---

Concrete Grammar Example II

Grammar Example II

---

Concrete Grammar Example II

\[
\begin{align*}
\text{Program} & ::= \text{program } \text{ident} ; \text{DeclSeq} \text{ begin } \text{StatSeq} \text{ end } \\
\text{DeclSeq} & ::= \text{Decl} ; \text{DeclSeq} | \epsilon \\
\text{Decl} & ::= \text{var } \text{ident} : \text{ident} \\
\text{Stat} & ::= \text{ident} ::= \text{Expr} | \text{if } \text{Expr} \text{ then } \text{StatSeq} \text{ else } \text{StatSeq} \\
\text{StatSeq} & ::= \text{Stat} ; \text{StatSeq} | \epsilon \\
\text{Expr} & ::= \text{ident} | \text{const} \\
\end{align*}
\]

Example: 

PROGRAM P;

VAR I : INTEGER;
VAR C : CHAR;
VAR J : INTEGER;
BEGIN I := 6; J := I; END.
Abstract Grammar...

Some items in the grammar are **attributes** (names of identifiers, e.g.) some are **children** (expression & statements in an if-statement, e.g.).

Every child & attribute in the abstract grammar is given a name:

```
LOP: Expr.
```

Example:

```
IfStat ::= Expr:Expr Then:Stat Else:Stat
```

**Input attributes** are data (e.g. identifiers, constants) created by the lexer/parser. I write them:

```
←Name:String.
```

Example:

```
IntConst ::= ←Value:INTEGER ←Pos:Position
```

I prefer linked lists to recursion to define lists. A statement sequence are statements linked on a child `Next:StatSeq`. Lists end with an empty node: `NoDecl`.

Grammar Example...

```
PROGRAM P;
VAR I : INTEGER;
VAR J : INTEGER;
VAR C : CHAR;
BEGIN
  I := 6;
  J := I;
END.
```

```
AST

Program
  Name="P"
  Pos=[1,1]
  DeclSeq
    Name="I"
    Pos=[2,8]
    TypeName="INTEGER"
    Next
    Name="J"
    Pos=[3,8]
    TypeName="INTEGER"
    Next
    Name="C"
    Pos=[4,8]
    TypeName="CHAR"
    Next
    NoDecl
  StatSeq
    Pos=[5,8]
    Stat
      Assign
        Name="I"
        Pos=[6,8]
        Expr
          Name="I"
          Pos=[7,6]
          Next
            TypeName="INTEGER"
            Pos=[8,6]
        Expr
      Next
        Name="J"
        Pos=[9,8]
        Next
          TypeName="INTEGER"
          Pos=[10,8]
          NoStat
      NoStat
      Name="I"
      Pos=[11,8]
      Next
    NoDecl
  Stat
    Assign
      Name="J"
      Pos=[12,8]
      Expr
        Value=6
```

```
Grammar Example III

Assign ::= ident := Expr
Expr ::= Expr + Term | Term
Term ::= Term * Factor | Factor
Factor ::= ( Expr ) | ident | const

Abstract Grammar (A):

Assign ::= Des: Name Expr: Expr ⇐ Pos: Position
Expr ::= Add | Mul | Name | IntConst
Add ::= LOP: Expr ROP: Expr ⇐ Pos: Position
Mul ::= LOP: Expr ROP: Expr ⇐ Pos: Position
Name ::= ⇐ Name: String ⇐ Pos: Position
IntConst ::= ⇐ Value: INTEGER ⇐ Pos: Position

Grammar Example III...

There is often more than way to design the abstract grammar.

We can turn attributes into node-kinds and vice versa.

Abstract Grammar (B):

Assign ::= Des: Name Expr: Expr ⇐ Pos: Position
Expr ::= Add | Mul | Name | IntConst
Add ::= LOP: Expr ROP: Expr ⇐ Pos: Position
Mul ::= LOP: Expr ROP: Expr ⇐ Pos: Position
Name ::= ⇐ Name: String ⇐ Pos: Position
IntConst ::= ⇐ Value: INTEGER ⇐ Pos: Position

I := J * 5 * (K + 3)
Regular languages are less powerful than context free languages.
Languages are organized in the Chomsky Hierarchy according to their generative power.
Type 3 languages are more restrictive (can describe simpler languages than) type 2 languages.
Type 3 languages can be parsed in linear time, type 2 languages in cubic time.
Programming languages are in between type 2 and 3.
Two natural languages (Swiss German and Bambara) are known not to be context free.
Chomsky’s theory suggests that every human utterance has two structures: surface structure, the superficial combining of words, and "deep structure," which are universal rules and mechanisms. In more practical terms, the theory argues that the means for acquiring a language is innate in all humans and is triggered as soon as an infant begins to learn the basics of a language. Outside this highly rarefied sphere, Chomsky early on began to promote his radical critique of American political, social, and economic policies, particularly of American foreign policy as effected by the Establishment and presented by the media; he was outspoken in his opposition to the Vietnam War and later to the Persian Gulf War. His extensive writings in this area include American Power and the New Mandarins (1969) and Human Rights and American Foreign Policy (1978).

Chomsky vs B. F. Skinner: Famous debate in the late 50’s, early 60’s. Skinner was a behaviorist, believing that children learn language by imitating their parents. Chomsky refuted this, claiming that we all have innate language mechanisms.

Nim Chimpsky was taught sign language in 1970s. It was a lost cause. He could ask for things, but not much more.
The job of a parser is to convert from concrete syntax to abstract syntax.
We use context free grammars to describe both the concrete and the abstract syntax.
The concrete syntax is described in the language manual of the language we're compiling.
The abstract syntax we make up ourselves. There are many ways to define the abstract syntax of a language and personal preference will play a role in how we construct it.

Use this abstract syntax to draw an AST for the Tiny program below:

```
BEGIN
  INT x;
  PRINT x + 9.9;
END
```