Programming Language Syntax — Topics

This is a BIG subject. See Sebesta Chapter 3.

- Grammars, BNF
- Derivation and Parsing
- Parse trees, Abstract Syntax Trees.
- Ambiguous grammars
- Encoding operator precedence in a grammar
- Associativity
- Extended BNF
- Syntax diagrams
Describing programming languages

Why do we need a description?

- Designers: to communicate ideas
- Implementors: to build a conforming compiler
- Programmers: how do I write a while loop in language X?

What kind of features can be described?

- **syntax**: legal sentences
- **semantics**
  - static: checked at compile time
  - dynamic: run time behaviour

We need a simple, compact and unambiguous notation:

☞ *We need a formal description rather than an informal (natural) language description.*
Formal syntax concepts

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabet</td>
<td>e.g. a–z, 0–9 etc</td>
</tr>
<tr>
<td>lexeme</td>
<td>smallest element (as returned by lexical analyser)</td>
</tr>
<tr>
<td>token</td>
<td>class of lexeme (e.g. identifier)</td>
</tr>
<tr>
<td>sentence</td>
<td>legal string of tokens</td>
</tr>
<tr>
<td>recogniser</td>
<td>:: (grammar (\times) sentence) (\rightarrow) Bool</td>
</tr>
<tr>
<td>generator</td>
<td>:: grammar (\rightarrow) [sentence]</td>
</tr>
</tbody>
</table>

- given a grammar, it is possible to automatically create a recogniser (parser)
- given a grammar, it is possible to automatically generate legal strings of a language. For most grammars, there are an infinite number of legal strings (programs).
- Warning: in practice token is often used as synonym for lexeme
Grammars – a first example

Consider a C declaration like:

void fun(int x, float y)

A simplified grammar for formal parameters. (e.g. int x, float y)

\[
\begin{align*}
params & \rightarrow \text{parameter} \\
params & \rightarrow \text{parameter} \ , \ params \\
\text{parameter} & \rightarrow \text{type} \ \text{identifier} \\
\text{type} & \rightarrow \text{int} \ | \ \text{float}
\end{align*}
\]

Derivation

\[
\begin{align*}
\text{params} & \Rightarrow \text{parameter} \ , \ \text{params} \\
& \Rightarrow \text{type} \ \text{identifier} \ , \ \text{params} \\
& \Rightarrow \text{int} \ \text{identifier} \ , \ \text{params} \\
& \Rightarrow \text{int} \ \text{identifier} \ , \ \text{parameter} \\
& \Rightarrow \text{int} \ \text{identifier} \ , \ \text{type} \ \text{identifier} \\
& \Rightarrow \text{int} \ \text{identifier} \ , \ \text{float} \ \text{identifier}
\end{align*}
\]

Note difference between → and ⇒
Grammars

A grammar is a set of rules describing how legal sentences of a language can be formed. A grammar has 4 elements

1. A set of nonterminal symbols.
   Non-terminal symbols typically denote phrases or sub-components of the language. (e.g. while-statement)

2. A set of terminal symbols.
   A legal sentence will contain only terminal symbols. (e.g. identifier, else)
   All terminal symbols are lexemes, and vice versa
   terminals and nonterminals are disjoint

3. A set of productions.
   A production is a rule which describes how to replace a string of symbols with another string.

4. A distinguished nonterminal or start symbol.
   By convention: the LHS of the first rule.
Derivation algorithm

This is the way that a sentence is generated from a grammar.

1. begin with $sentence = \text{start symbol}$
2. repeat until only terminals remain in $sentence$:
   replace one occurrence of the symbol(s) in $sentence$ matching the LHS of a production with the RHS of that production

Note:

- there may be many occurrences of a single production’s LHS in $sentence$
  – this can sometimes cause problems (see later—ambiguity)
- there may be many possible instances of a symbol sequence matching a production LHS in $sentence$
  – this is not problematical
- in general, an infinite number of legal sentences may be produced
Productions & derivations

production: \( AB \rightarrow CDE \)

derivation: \( \textit{start-symbol} \)
\[ \Rightarrow \ldots \]
\[ \Rightarrow EFABXY \]
\[ \Rightarrow EFCD\text{DEXY} \]
\[ \Rightarrow \ldots \]

Notes

1. often terminals and non-terminals are written differently to help distinguish them.
   
   E.g. lower case for terminals, upper case for non-terminals.

2. A grammar in general describes how to produce an infinite number of correct sentences. Only a small proportion will be meaningful.
# Example 1—A regular grammar

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Possible derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow A$</td>
<td>$S \Rightarrow A$</td>
</tr>
<tr>
<td>$A \rightarrow aA$</td>
<td>$\Rightarrow aA$</td>
</tr>
<tr>
<td>$A \rightarrow aB$</td>
<td>$\Rightarrow aaB$</td>
</tr>
<tr>
<td>$B \rightarrow bB$</td>
<td>$\Rightarrow aabB$</td>
</tr>
<tr>
<td>$B \rightarrow b$</td>
<td>$\Rightarrow aabb$</td>
</tr>
</tbody>
</table>

## Notes

1. This grammar generates sentences of the form: $a^m b^n$, $m \geq 1$, $n \geq 1$
2. Limitation: cannot reliably form sentences: $a^m b^n$, where $m = n$
3. This example is from the class of regular grammars that can consist of only these productions:
   - $A \rightarrow xB$ or $A \rightarrow x$ (or $A \rightarrow Bx$ or $A \rightarrow x$)
   - $A, B \in \text{nonterminal}; x \in (\text{terminal, } \epsilon)$
Example 2—A context-free grammar

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Possible derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow ASB$</td>
<td>$S \Rightarrow ASB$</td>
</tr>
<tr>
<td>$S \rightarrow ab$</td>
<td>$S \Rightarrow AASBB$</td>
</tr>
<tr>
<td>$A \rightarrow a$</td>
<td>$S \Rightarrow AAabBB$</td>
</tr>
<tr>
<td>$B \rightarrow b$</td>
<td>$A \rightarrow aAabBB$</td>
</tr>
<tr>
<td>$A \rightarrow a$</td>
<td>$A \rightarrow aaabBB$</td>
</tr>
<tr>
<td>$B \rightarrow b$</td>
<td>$B \rightarrow aaabbB$</td>
</tr>
<tr>
<td></td>
<td>$B \rightarrow aaabbb$</td>
</tr>
</tbody>
</table>

Notes

1. This grammar generates sentences of the form: $a^n b^n, n \geq 1$
2. CFGs are more powerful than regular grammars: same number of $a$ as $b$
3. Context-free grammar rules have this form:
   - $A \rightarrow x_1 \ldots x_n$, where $n \geq 0$ (if $n = 0$, empty production: $A \rightarrow \epsilon$)
   - $A \in \text{nonterminal}; x_i \in (\text{nonterminal, terminal})$
### Example 3—A context sensitive grammar

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Possible derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow aSBC$</td>
<td>$S \Rightarrow aSBC$</td>
</tr>
<tr>
<td>$S \rightarrow abC$</td>
<td>$\Rightarrow aabCBC$</td>
</tr>
<tr>
<td>$CB \rightarrow BC$</td>
<td>$\Rightarrow aabBC'C$</td>
</tr>
<tr>
<td>$bB \rightarrow bb$</td>
<td>$\Rightarrow aabbCC'$</td>
</tr>
<tr>
<td>$bC' \rightarrow bc$</td>
<td>$\Rightarrow aabbc'C$</td>
</tr>
<tr>
<td>$cC' \rightarrow cc$</td>
<td>$\Rightarrow aabbCC$</td>
</tr>
</tbody>
</table>

**Notes**

1. This grammar generates sentences of the form: $a^n b^n c^n, n \geq 1$

2. There can be $> 1$ symbols on LHS of the production: expansion of $C$ above depends on its context (what surrounds it in the sentence).

3. Context-sensitive grammar rules have this form:
   
   $$x_1 \ldots x_n \rightarrow y_1 \ldots y_m, \text{ where } n \geq 1$$

   $$n \leq m \land x_i, y_i \in (\text{nonterminal, terminal})$$
More notation

• grouping RHS of productions

\[ B \rightarrow bB \mid b \]

is equivalent to the two productions

\[ B \rightarrow bB \]
\[ B \rightarrow b \]

This is done to make writing rules simpler and clearer.

• the empty production

The production with zero RHS symbols, \( A \rightarrow \) is usually written with an explicit empty symbol: \( A \rightarrow \epsilon \)

For example: \( B \rightarrow bB \mid \epsilon \) (zero or more occurrences)
Kinds of grammars

Chomsky Hierarchy of Formal Languages

Regular $\subset$ Context-free $\subset$ Context-sensitive $\subset$ Unrestricted

Regular grammar

- used to describe the lexemes of a language
- used to construct lexical analyzers (scanner)
- tools are available to generate a lexical analyzer from a grammar (e.g. Lex)

Context-free Grammar

- used to describe the syntax of a language
- used to construct syntax analyzers (parser)
- tools are available to generate a syntax analyzer from a grammar (e.g. Yacc, Bison, PCCTS)
Backus-Naur Form

• BNF describes *context-free grammars*

• BNF: Backus-Naur Form (sometimes: Backus normal form)
  – John Backus: inventor for Fortran
    invented a notation to describe Algol 58
  – Peter Naur
    extended the notations for Algol 60

• developed independantly of Chomsky

• used universally for programming language syntax description
BNF elements

1. **syntactic symbols** (non-terminals)
   - abstraction of various syntax structures e.g. *statement*, *expression*
   - usually written as *italic* or *<bracketed>*

2. **tokens** (terminals)
   - language elements returned by lexer e.g. *if*, *var*
   - conventions
     - reserved words & symbols: *if* or “if”, := or “:=”, , (),
     - others: *var* or *<var>*

3. **rules** (productions)
   - LHS: a syntactic symbol
   - RHS: syntax structure of the symbol

4. **start symbol** — usually *program*
A “Real” Example

Grammar 3.1 from Sebesta:

\[
\begin{align*}
program & \rightarrow \text{begin stmt_list end} \\
stmt_list & \rightarrow stmt \mid stmt \; ; \; stmt_list \\
stmt & \rightarrow var \; := \; expression \\
var & \rightarrow A \mid B \mid C \\
expression & \rightarrow var \; + \; var \\
 & \mid \; var \; - \; var \\
 & \mid \; var
\end{align*}
\]
Rightmost derivation of \( \text{begin A := B+C; B := C end} \)

\[
\begin{align*}
\text{program} & \Rightarrow \text{begin stmt_list end} \\
& \Rightarrow \text{begin stmt ; stmt_list end} \\
& \Rightarrow \text{begin stmt ; stmt end} \\
& \Rightarrow \text{begin stmt ; var := expression end} \\
& \Rightarrow \text{begin stmt ; var := var end} \\
& \Rightarrow \text{begin stmt ; var := C end} \\
& \Rightarrow \text{begin stmt ; B := C end} \\
& \Rightarrow \text{begin var := expression ; B := C end} \\
& \Rightarrow \text{begin var := var + var ; B := C end} \\
& \Rightarrow \text{begin var := var + C ; B := C end} \\
& \Rightarrow \text{begin var := B + C ; B := C end} \\
& \Rightarrow \text{begin A := B + C ; B := C end}
\end{align*}
\]
Derivation ....

- Compare with Sebesta leftmost derivation

- derivation order can be
  - leftmost
  - rightmost
  - random

- any order gives same sentence
  (in this case; because the grammar is unambiguous)

- final line contains only terminals

- intermediate lines are called sentential forms
BNF idioms

**Sequence**  \[ A \rightarrow B \; C \; D \]

while-statement \[ \rightarrow \text{while ( expression ) statement} \]

**Alternation**  \[ C \rightarrow X \; | \; Y \]

unary-operator \[ \rightarrow \& \; | \; * \; | \; + \; | \; - \; | \; ~ \; | \; ! \]

**List** of \( A \) with optional separator \( S \)

\[
\begin{array}{|c|c|}
\hline
\text{≥ 1 elements, no separator} & L \rightarrow A \; | \; AL \\
\hline
\text{≥ 0 elements, no separator:} & L \rightarrow AL \; | \; \epsilon \\
\hline
\text{≥ 1 elements, with separator} & L \rightarrow A \; | \; ASL \\
\hline
\text{≥ 0 elements, with separator} & L_0 \rightarrow L \; | \; \epsilon \\
\hline
& L \rightarrow A \; | \; ASL \\
\hline
\end{array}
\]

examples (from C)

\[
\begin{align*}
\text{stmt\_list} & \rightarrow \text{stmt} \; | \; \text{stmt} \; \text{stmt\_list} \\
\text{stmt\_list}_0 & \rightarrow \text{stmt\_list} \; | \; \epsilon \\
\text{var\_list} & \rightarrow \text{var} \; | \; \text{var} \; , \; \text{var\_list}
\end{align*}
\]
Parse trees

Consider the grammar:

\[
\begin{align*}
assign & \rightarrow id := expr \\
id & \rightarrow A | B | C \\
expr & \rightarrow id + expr | id * expr | ( expr ) | id
\end{align*}
\]

Derivation of \( A := B * ( A + C ) \):

\[
\begin{align*}
assign & \Rightarrow id := expr \\
& \Rightarrow A := expr \\
& \Rightarrow A := id * expr \\
& \Rightarrow A := B * expr \\
& \Rightarrow A := B * ( expr ) \\
& \Rightarrow A := B * ( id + expr ) \\
& \Rightarrow A := B * ( A + expr ) \\
& \Rightarrow A := B * ( A + id ) \\
& \Rightarrow A := B * ( A + C )
\end{align*}
\]
A **parse tree** is the graphical description of a derivation

- nodes are nonterminals
- leaves are terminals
- root is the start symbol
- each derivation step expands a node
- tree shape is independent of derivation order for unambiguous grammar (see later)
Abstract syntax tree

Example 2: fun(a+b, -1)

\[
\begin{array}{c}
\text{fun} \quad + \quad - \\
\text{a} \quad b \quad 1
\end{array}
\]

- removes:
  - unnecessary detail (abstraction)
  - parsing info (nonterminals)
  - punctuation used in grammar (e.g., ; [ ] ( ))

- only terminals are present

- nodes are basic semantic operations (can be: unary, binary, n-ary)
Ambiguous grammars

Definition: A grammar is ambiguous if a sentence exists for which more than one parse tree can be constructed.

Ambiguous grammar

\[
\begin{align*}
assign & \rightarrow id := expr \\
id & \rightarrow A \mid B \mid C \\
expr & \rightarrow expr + expr \mid expr * expr \mid ( expr ) \mid id
\end{align*}
\]

Consider derivation of \( A := A + B * C \)

```
=  /
\  / \  \\
/   /   /   \\
A   +   A
   /   /   \\
  /   /   \  \\
 /   /   +   \\
/   /   /   \ \
B   C   A   B
```
An unambiguous grammar

```
expr  →  id + expr | id * expr | ( expr ) | id
```

- the unambiguous grammar is more restrictive
  E.g.: can’t write:  \((A+B) \times (C+B)\)

- the unambiguous grammar still allows weird precedence (but at least it is not ambiguous!)
  e.g. \(A := A + B \times C + D \equiv A := A + (B \times (C + D))\)

- we can write grammars to create correct precedence ...
Encoding Operator Precedence in CFG

- expressions lower in the AST are evaluated first
- so: ensure higher-precedence operators appear lower in parse tree
- how?: introduce new nonterminals to generate same-level operations

```
assign  →  id := expr
id      →  A | B | C
expr    →  expr + term | term
term    →  term * factor | factor
factor  →  ( expr ) | id
```

- \( expr \Rightarrow^* term + term + term + \cdots \)
- \( term \Rightarrow^* factor * factor * factor * \cdots \)
- the \emph{term} is always evaluated \textbf{before} summing to give expression value
- this grammar is unambiguous because it leads to a limited choice of which non-terminal to select during derivation.
Unambiguous derivation of $A + B + C$

\[
expr \Rightarrow expr + term \\
\Rightarrow expr + term + term \\
\Rightarrow term + term + term
\]

- there is no choice of $expr$ to expand so get a unique parse tree structure.

Compare the above with the ambiguous

\[
expr \rightarrow expr + expr | term
\]

- ambiguous derivation of $A + B + C$
  \[
  expr \Rightarrow expr + expr \Rightarrow expr + expr + expr \\
  expr \Rightarrow expr + expr \Rightarrow expr + expr + expr
  \]

- these give different parse trees

```
    e
   /\  \
  e + e
 /\  \
/\  \
|  \
|  e + e
```

```
    e
   /\  \
  e + e
 /\  \
/\  \
|  \
|  e + e
```
Associativity

• Is $a \oplus b \oplus c$ equivalent to $(a \oplus b) \oplus c$ or $a \oplus (b \oplus c)$

• why does it matter?
  – some operators are not associative.
    e.g. division and exponentiation (try $2/3/2$ using left & right assoc.)
  – floating point arithmetic
    e.g. imagine only 4 digits of accuracy
    left association: $(5+9)+10000 = 14 + 10000 = 10014 = 10010$
    right association: $5+(9+10000) = 5 + 10009 = 5 + 10000 = 10005$
    $= 10000$

• Encoding Associativity with a CFG
  – left recursion ⇔ left association
    $\text{list} \rightarrow \text{list item} \mid \text{item}$
  – right recursion ⇔ right association
    $\text{list} \rightarrow \text{item list} \mid \text{item}$
Ambiguity revisited: The if-then-else problem

• consider a simple grammar for an if statement

\[
\text{if\_stmt} \rightarrow \text{if} \ \text{cond} \ \text{then} \ \text{stmt} \\
| \ \text{if} \ \text{cond} \ \text{then} \ \text{stmt} \ \text{else} \ \text{stmt}
\]

• Now look at the if..if statement

\[
\text{if} \ c1 \ \text{then if} \ c2 \ \text{then} \ s1 \ \text{else} \ s2
\]

• Which “if” does \text{s2} belong to?

\[
\text{<if>}<\text{if}> \\
/\/ \| \| \\
/\ \| \| \\
c1<\text{if}>c1<\text{if}>s2 \\
/\|\| /\|\| \\
c2\ s1\ s2\ c2\ s1
\]
The if-then-else problem—solved

Solution is to disallow an unmatched if in the “then” part of a matched if.

\[
\begin{align*}
\text{stmt} & \rightarrow \text{match} \mid \text{unmatch} \\
\text{match} & \rightarrow \text{if cond then match else match} \\
& \quad \mid \text{any\_other\_stmt} \\
\text{unmatch} & \rightarrow \text{if cond then stmt} \\
& \quad \mid \text{if cond then match else unmatch}
\end{align*}
\]
Extended BNF

Extra meta-syntax can make BNF grammars simpler

| Universal: | [ ... ] | optional |
|           | { ... } | 0 or more |
|           | ( ... | ... ) | alternation |

| Sometimes: | object* | 0 or more |
|           | object+ | 1 or more |

- \( \text{varDecl} \rightarrow \text{typeName} \ \text{var} \ \{, \ \text{var}\}^{*} \)
- \( \text{block} \rightarrow \{\} \ \text{stmt}^{+} \ \} \)
- \( \text{procCall} \rightarrow \text{ident} \ (\ [\text{exprList}] ) \)
- \( \text{expr} \rightarrow \text{term} \ \{(\text{+|\text{\-}}) \ \text{term}\} \)
- Note use of boxes to distinguish concrete and meta syntax
Syntax Graphs

unsigned integer

digit

unsigned number

unsigned integer

digit

E

unsigned integer

+}

−
Syntax Graph Syntax

syntaxGraph → name

empty
non terminal
terminal
sequence
iteration
alternation
# Syntax Graphs Idioms

<table>
<thead>
<tr>
<th>Alternation</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Alternation Diagram" /></td>
<td><img src="image" alt="Iteration Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternation</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \neq B \Rightarrow (A</td>
<td>B) )</td>
</tr>
<tr>
<td>( A \equiv \epsilon \Rightarrow [B] )</td>
<td>optional</td>
</tr>
<tr>
<td>( B \equiv \epsilon \Rightarrow [A] )</td>
<td>optional</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \neq B \Rightarrow A{BA}^* \geq 1, \text{ separator} )</td>
</tr>
<tr>
<td>( B \equiv \epsilon \Rightarrow A^+ \geq 1, \text{ no separator} )</td>
</tr>
<tr>
<td>( A \equiv \epsilon \Rightarrow B^* \geq 0, \text{ no separator} )</td>
</tr>
</tbody>
</table>