CSc 520

Principles of Programming Languages

51: Semantics — Syntax

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg

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:

*IfStat → if ( expr ) 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 for English

A grammar can be used for

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

Let’s look at a simple grammar for a fragment of English.

Syntactic Categories

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

A Simple English Grammar

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S ↔ NP VP</td>
<td>NP VP</td>
</tr>
<tr>
<td>NP ↔ N</td>
<td>N VP</td>
</tr>
<tr>
<td>N ↔ John</td>
<td>John VP</td>
</tr>
<tr>
<td>VP ↔ V NP</td>
<td>John V NP</td>
</tr>
<tr>
<td>V ↔ kissed</td>
<td>John kissed NP</td>
</tr>
<tr>
<td>NP ↔ N</td>
<td>John kissed N</td>
</tr>
<tr>
<td>N ↔ Lisa</td>
<td>John kissed Lisa</td>
</tr>
<tr>
<td>N ↔ John</td>
<td></td>
</tr>
<tr>
<td>N ↔ Lisa</td>
<td></td>
</tr>
<tr>
<td>V ↔ house</td>
<td></td>
</tr>
<tr>
<td>V ↔ died</td>
<td></td>
</tr>
<tr>
<td>V ↔ kissed</td>
<td></td>
</tr>
<tr>
<td>Det ↔ the</td>
<td></td>
</tr>
<tr>
<td>Det ↔ a</td>
<td></td>
</tr>
</tbody>
</table>

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

Sentence Generation

1. Start with the start symbol.
2. Pick a non-terminal \( X \) on the right hand side.
3. Pick a grammar rule \( X \rightarrow \gamma \).
4. Replace \( X \) with \( \gamma \).
5. Repeat until left with a string of words.

Terminology

A grammar is a 4-tuple

\((N, \Sigma, P, S)\)

or

\((non-terminals, terminals, productions, start-symbol)\)

A production is of the form \( \alpha \rightarrow \beta \) where \( \alpha, \beta \) are taken from \( N \cup \Sigma \).
Read \( \alpha \rightarrow \beta \) as “rewrite \( \alpha \) with \( \beta \)”.
Read \( \Rightarrow \) as “directly derives”.
Read \( \Rightarrow^r \) as “directly derives using rule \( r \)”.
Read \( \Rightarrow^* \) as “derives in one or more steps”.

A grammar is a 4-tuple

\((N, \Sigma, P, S)\)

or

\((non-terminals, terminals, productions, start-symbol)\)

A production is of the form \( \alpha \rightarrow \beta \) where \( \alpha, \beta \) are taken from \( N \cup \Sigma \).
Read \( \alpha \rightarrow \beta \) as “rewrite \( \alpha \) with \( \beta \)”.
Read \( \Rightarrow \) as “directly derives”.
Read \( \Rightarrow^r \) as “directly derives using rule \( r \)”.
Read \( \Rightarrow^* \) as “derives in one or more steps”.

S, NP, VP, N, Det, V are non-terminal symbols.
John, Lisa, house, died, ... are terminal symbols.
S is the start symbol.
Here’s a grammar for a simple programming language:

\[
\text{Program ::= BEGIN Stat END}
\]

\[
\text{Stat ::= ident ::= Expr}
\]

\[
\text{Expr ::= Expr + Expr | Expr * Expr | ident | number}
\]

We write terminal symbols like this. We write non-terminal symbols like this. Sometimes we write ::= instead of →.

\[A \rightarrow b \mid c\] is the same as \[A \rightarrow b; A \rightarrow c\]. Read | as “or”.

We know the sentence

\[
\text{BEGIN a ::= 5 + 4 * 3 END}
\]

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

\[
\begin{align*}
\text{Program} & \Rightarrow \text{BEGIN Stat END} \\
& \Rightarrow \text{BEGIN ident ::= Expr END} \\
& \Rightarrow \text{BEGIN "a" ::= Expr END} \\
& \Rightarrow \text{BEGIN "a" ::= 5 + Expr END} \\
& \Rightarrow \text{BEGIN "a" ::= 5 + Expr * Expr END} \\
& \Rightarrow \text{BEGIN "a" ::= 5 + 4 * Expr END} \\
& \Rightarrow \text{BEGIN "a" ::= 5 + 4 * 3 END}
\end{align*}
\]

Our English grammar is the 4-tuple

\[
\{\{S, NP, V, \ldots\}, \\
\{\text{John, house, died, \ldots}\}, \\
\{S \rightarrow NP VP, VP \rightarrow V, \ldots\}, \\
S\}
\]

Our PL grammar is the 4-tuple

\[
\{\{\text{Program, Stat, \ldots}\}, \\
\{\text{BEGIN, ::=, *, \ldots}\}, \\
\{\text{Program ::= BEGIN Stat END, \ldots}\}, \\
\text{Program}\}
\]

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

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

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

```
A
  /\  \
 X Y Z
```

---

**Parse Trees...**

---

**Parser Trees...**

---

**Parse Trees...**

---

**Parse Trees...**

---

**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.
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 { ... }
```

In BNF, our variable declaration

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

looks like this:

```java
vars ::= ident ident idlist |
idlist ::= , ident idlist | e
```

In EBNF, it looks like this:

```java
vars ::= ident ident { , ident } |
```

I.e. \{e\} means that e is repeated 0 or more times.

EBNF...

In BNF, our class declaration

```java
class C extends D { ... }
```

looks like this:

```java
class ::= class ident extends { ... }
extends ::= extends ident | e
```

In EBNF, it looks like this:

```java
class ::= class ident [extends ident] { ... }
```

I.e. [e] means that e is optional.
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 ] ENDIF |
FOR ident ::= expression TO expression [ BY expression ] DO
stat_seq ENDFOR |
WHILE expression DO stat_seq ENDDO |
REPEAT stat_seq UNTIL expression |
LOOP stat_seq ENDOFF | EXIT

EBNF for Luca...

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

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

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.

\[ E ::= E + E \mid E * E \mid \text{number} \]

\[ 5 + 4 * 3 \]

Structural Ambiguity in English

Ambiguities occur in natural languages also:

\[ \text{I saw the man with binoculars} \]

\[ \text{I saw the } \text{man with binoculars} \]

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 \(+\):

\[ 4 + 5 * 3 \]

means

\[ 4 + (5 * 3), \]

not

\[ (4 + 5) * 3. \]

We say that \(*\) binds harder than \(+\).

Operator Associativity

The associativity of an operator describes how operators of equal precedence are grouped.

\[ + \text{ and } - \text{ are usually left associative:} \]

\[ 4 - 2 + 3 \]

means

\[ (4 - 2) + 3 = 5, \]

not

\[ 4 - (2 + 3) = -1. \]

We say that \(+\) associates to the left.

\[ \text{\^} \text{ associates to the right:} \]

\[ 2 \text{\^} 3 \text{\^} 4 = 2 \text{\^} (3 \text{\^} 4). \]
# Expression Grammars

- We must write unambiguous expression grammars that reflect the associativity and precedence of all operators.

- The next slide gives the algorithm for writing such grammars.

## Resulting Expression Grammar:

$\text{expr ::= expr} \pm \text{term} \mid \text{term}$

$\text{term ::= term} \ast \text{factor} \mid \text{factor}$

$\text{factor ::= ( expr )} \mid \text{number}$

---

## Expression Grammars...

1. Create one non-terminal for each precedence level, for example $p_1, p_2, \cdots, p_n$, where $p_n$ has the highest precedence level.

2. For operator $\text{op}$ at precedence level $i$ construct the following production if the operator is:
   - left associative:
     
     $p_i ::= p_i \text{ op } p_{i+1} \mid p_{i+1}$

   - right associative:
     
     $p_i ::= p_{i+1} \text{ op } p_i \mid p_{i+1}$

3. Construct a production for nonterminal $p_{n+1}$ which represents primary expressions such as identifiers, numbers, parenthesized expressions, etc:

   $p_{n+1} ::= (p_1) \mid \text{num} \mid \text{id}$

---

<table>
<thead>
<tr>
<th>Operator</th>
<th>Kind</th>
<th>Prec</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\cdot$, $\ast$, $%$</td>
<td>Binary</td>
<td>13</td>
<td>Left</td>
</tr>
<tr>
<td>$\pm$, $-$</td>
<td>Binary</td>
<td>12</td>
<td>Left</td>
</tr>
<tr>
<td>$&lt;&lt;$, $&gt;&gt;$</td>
<td>Binary</td>
<td>11</td>
<td>Left</td>
</tr>
<tr>
<td>$&lt;$, $&gt;$, $&lt;=$, $=&gt;$</td>
<td>Binary</td>
<td>10</td>
<td>Left</td>
</tr>
<tr>
<td>$==$, $!=$</td>
<td>Binary</td>
<td>9</td>
<td>Left</td>
</tr>
<tr>
<td>$&amp;$</td>
<td>Binary</td>
<td>8</td>
<td>Left</td>
</tr>
<tr>
<td>$\hat{\ast}$</td>
<td>Binary</td>
<td>7</td>
<td>Left</td>
</tr>
<tr>
<td>$&amp;$, $\rangle$</td>
<td>Binary</td>
<td>6</td>
<td>Left</td>
</tr>
<tr>
<td>$</td>
<td>$, $\langle$</td>
<td>Binary</td>
<td>5</td>
</tr>
<tr>
<td>$</td>
<td>$, $</td>
<td>$</td>
<td>Binary</td>
</tr>
<tr>
<td>$?:$</td>
<td>Ternary</td>
<td>3</td>
<td>Right</td>
</tr>
<tr>
<td>$\times$, $\div$, $\ast$, $\div$, $\langle$, $\rangle$, $\langle$, $\rangle$</td>
<td>Binary</td>
<td>2</td>
<td>Right</td>
</tr>
<tr>
<td>$\ast$, $\div$, $\ast$, $\div$, $\langle$, $\rangle$, $\langle$, $\rangle$</td>
<td>Binary</td>
<td>1</td>
<td>Left</td>
</tr>
</tbody>
</table>

---

520—Spring 2005—51

520—Spring 2005—51

520—Spring 2005—51

520—Spring 2005—51
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.

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:

Assign ::= \text{ident} \ Expr
If ::= Expr StatSeq
IfElse ::= Expr StatSeq StatSeq
While ::= Expr StatSeq
Stat ::= Assign | If | IfElse | While
StatSeq ::= Stat StatSeq | NULL

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):

\[
\text{StatSeq} ::= \text{Stat} \text{ StatSeq} | \text{NULL}
\]
Grammar Example I...

\[\text{Stat} ::= \text{Assign} | \text{If} | \text{IfElse} | \text{While}\]
\[\text{StatSeq} ::= \text{Stat} \text{StatSeq} | \text{NULL}\]

Concrete Grammar Example II

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

Example:

\[\text{PROGRAM P;\begin{align*}\text{VAR I} & : \text{INTEGER}; \\
\text{VAR C} & : \text{CHAR}; \\
\text{VAR J} & : \text{INTEGER}; \\
\text{BEGIN} & \text{I := 6; J := I; END.}\end{align*}\]

Abstract Grammar...

- Some items in the grammar are \textit{attributes} (names of identifiers, e.g.) some are \textit{children} (expression & statements in an if-statement, e.g.).
- Every child & attribute in the abstract grammar is given a name:
  \[\text{LOP} : \text{Expr.}\]
- Example:
  \[\text{IfStat} ::= \text{Expr} \text{Expr Then} : \text{Stat} \text{Else} : \text{Stat}\]

Abstract Grammar...

- Input attributes are data (e.g. identifiers, constants) created by the lexer/parser. I write them:
  \[\text{Name} : \text{String}.\]
- Example:
  \[\text{IntConst} ::= \text{Value} : \text{INTEGER} \text{Pos} : \text{Position}\]
- I prefer linked lists to recursion to define lists. A statement sequence are statements linked on a child
  \[\text{Next} : \text{StatSeq}.\]
  Lists end with an empty node: \text{NoDecl}.\]
Abstract Grammar:

Program ::= \langle Name: String \rangle \langle DeclSeq: Decl \rangle
StatSeq ::= \langle Pos: Position \rangle
Decl ::= VarDecl | FuncDecl | \cdots | NoDecl
VarDecl ::= \langle Name: String \rangle \langle TypeName: String \rangle \langle Pos: Position \rangle
NextVar
Stat ::= Assign | IfStat | \cdots | NoStat
Assign ::= Des: Name Expr: Expr \langle Pos: Position \rangle Next: Stat
IfStat ::= Expr: Expr Then: Stat Else: Stat \langle Pos: Position \rangle
NextStat
Expr ::= Name | IntConst
Name ::= \langle Name: String \rangle \langle Pos: Position \rangle
IntConst ::= \langle Value: INTEGER \rangle \langle Pos: Position \rangle

Grammar Example III

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

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 \langle Pos: Position \rangle
Expr ::= Add | Mul | Name | IntConst
Add ::= LOP: Expr ROP: Expr \langle Pos: Position \rangle
Mul ::= LOP: Expr ROP: Expr \langle Pos: Position \rangle
Name ::= \langle Name: String \rangle \langle Pos: Position \rangle
IntConst ::= \langle Value: INTEGER \rangle \langle Pos: Position \rangle
Grammar Example III...

I := J * 5 * (K + 3)

Compiler Grammars

The Chomsky Hierarchy

<table>
<thead>
<tr>
<th>TYPE</th>
<th>GRAMMAR</th>
<th>PSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unrestricted</td>
<td>$\alpha \rightarrow \beta$</td>
</tr>
<tr>
<td>1</td>
<td>Context Sensitive</td>
<td>$\alpha \rightarrow \beta$, $</td>
</tr>
<tr>
<td>2</td>
<td>Context Free</td>
<td>$A \rightarrow \beta$</td>
</tr>
<tr>
<td>3</td>
<td>Regular</td>
<td>$A \rightarrow a\beta$, $A \rightarrow a$</td>
</tr>
</tbody>
</table>
The Chomsky Hierarchy...

- 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.

Noam Chomsky

www.geocities.com/Athens/Acropolis/5148/chomskybio.html
Linguist, social/political theorist; born in Philadelphia. Son of a distinguished Hebrew scholar, he was educated at the University of Pennsylvania, where he was especially influenced by Zellig Harris; after taking his M.A. there in 1951, he spent four years as a junior fellow at Harvard (1951–55), then was awarded a Ph.D. from the University of Pennsylvania (1955). In 1955 he began what would be his long teaching career at the Massachusetts Institute of Technology. He became known as one of the principal founders of transformational-generative grammar, a system of linguistic analysis that challenges much traditional linguistics and has much to do with philosophy, logic, and psycholinguistics; his book Syntactic Structures (1957) was credited with revolutionizing the discipline of linguistics.

Noam Chomsky...

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).

Noam Chomsky...

- “If the Nuremberg laws were applied today, then every Post-War American president would have to be hanged.”
- “The corporatization of America during the past century [has been] an attack on democracy.”
- “Any dictator would admire the uniformity and obedience of the [U.S.] media.”
- “Judged in terms of the power, range, novelty and influence of his thought, Noam Chomsky is arguably the most important intellectual alive.” (The New York Times Book Review)
Noam Chomsky...

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.

Summary

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.

Readings and References

- Read Scott, Chapter 2: Programming Language Syntax
- Read Louden:
  - Regular Expressions 34–47.
  - Context-Free Grammars 95–142.
- or the Dragon Book:
  - grammars 165–171
  - associativity & precedence 30–32
  - ambiguity 171,174–175
  - derivations 167–169
  - parse trees 169–171
  - top-down parsing 41–43
  - left recursion 47–48

Exam Problem

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

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

```
PROGRAM    →  STATSEQ
STATSEQ    →  STAT STATSEQ | NULL
STAT       →  ASSIGN | PRINT | DECL
DECL       →  ident type
ASSIGN     →  ident EXPR
PRINT      →  EXPR
EXPR       →  BINOP | IDENT | INTLIT
BINOP      →  op EXPR EXPR
IDENT      →  ident
INTLIT     →  int
FLTLIT     →  fbat
```