Basic Components — Syntax

Steven Zeil

Sep. 7, 2001

Contents

1 Translation

2 Syntax

2.1 Separation of Form from Intent

2.1.1 Abstract Syntax

2.1.3 Concrete Syntax

2.2 Recursive Descent Parsing

2.3 Miscellaneous notes

Basic Components of Programming Languages

1. History

2. Classification

3. Translation

3 Translation

1. phases of Translation

2. Lexical: What are the words?

3. Syntax: What is the grammatical structure combining the words into sentences?

4. Semantics: What do the sentences mean?

2.1 Syntax

The syntax of a language is the set of rules describing how tokens can be combined to form sentences.

(“sentence” = “program”, for our purposes)

1. Separation of Form from Intent

2. Abstract Syntax

3. Concrete Syntax

2.1.1 Separation of Form from Intent

The same sentence can be expressed in many different ways by altering the syntactic rules.

Idea: Add A to the product of B and C, then divide the result by B.

How to express this?: The form we choose must capture such details as “the + operator is applied to A and to the result of the \( B \times C \) calculation.”

Some possibilities are

**Infix form:** Put each operator in between its operands.

**Prefix form:** Put each operator before its operands.

**Postfix form:** Put each operator after its operands.

Idea: Add A to the product of B and C, then divide the result by B.

Expressed as:

**Infix form:** \( (A + B \times C)/B \)

**Prefix form:** \( / + A \times B C B \)

**Postfix form:** \( ABC + B / \)
Basic Components — Syntax

Infix Notation

- most familiar
- actually more complicated than prefix or postfix

For example, how do we know to write
\[(A + B \times C)/B\]

instead of
\[A + B \times C/B\]

How do we know that we don’t need to write
\[(A + (B \times C))/B\]

We augment infix notation using associativity and precedence.

Associativity

Associativity is the rules by which sequences of the same operation are evaluated.

In conventional algebra,
- \(+, -, \text{ and } \times\) are left associative.
  \(4 - 2 - 1\) is grouped as 
  \((4-2)-1\)
- \(**\) (raises to the power) is right associative.
  \(4 \times 2 \times 3 = 4^{2^3}\) is grouped as \(4 \times (2 \times 3) = 4^{(2^3)}\).

Precedence

Precedence is the rules by which sequences of different operations are evaluated.

In conventional algebra, we have the following precedences

\[
\begin{array}{c|c|c|c}
** & \times, / & +, - \\
\hline
\text{higher} & \text{lower}
\end{array}
\]

So \(1 + 2 \times 3\) groups as \(1 + (2 \times 3)\).

Confusion often results because
- Programmers sometimes forget the algebraic rules.
- Some HLL’s have unexpected precedence and associativity rules.

Confusion often results because
- Programmers sometimes forget the algebraic rules.
- Some HLL’s have unexpected precedence and associativity rules.
  - APL gave all operators the same precedence and right associativity.

Confusion often results because
- Programmers sometimes forget the algebraic rules.
- Some HLL’s have unexpected precedence and associativity rules.
  - Some languages give unary \(-\) the highest precedence; some give it the lowest.
    What is \(-x * -y\)?
    How about \(-2 * -1\)?

Confusion often results because
- Programmers sometimes forget the algebraic rules.
- Some HLL’s have unexpected precedence and associativity rules.
  - Pascal boolean ops
    \[
    \text{if } A < B \text{ and } C > D \text{ then }
    \]
    groups as
    \[
    A < (B \text{ and } C) > D
    \]
    causing a syntax error.

Confusion often results because
- Programmers sometimes forget the algebraic rules.
  - esp. right-assoc. of **

Confusion often results because
- In C, if \(p\) is a pointer to a personnel record, then \((*p)\).salary gets the salary. What about \(*p\).salary?
2.1.2 Abstract Syntax

So far, we have been pre-occupied with

- how tokens group together
- which operators apply to which operands

We call these properties the **abstract syntax** of the language.

---

**Abstract Syntax Trees**

Grouping and application of operators are easily illustrated using trees.

For example, our interpretation of the infix

\[ x \times (y + 1) - z \]

is

![Abstract Syntax Tree](image)

Such **abstract syntax trees** (AST's) are interpreted as

- internal nodes:
  - labelled with tokens denoting operators
  - children are the operands
- leaves contain non-operator tokens (e.g., variables, constants)

Note that the () do not appear in the AST. These are “syntactic sugar” — they don’t matter once we have grasped the appropriate grouping and application structure.

---

2.1.3 Concrete Syntax

The **concrete syntax** of a language is the set of rules governing how it is actually written.

- generally described using grammars
  - for HLL's, context-free grammars
A **language** is a set of strings.  
A **grammar** $G = [T, N, P, S]$ is a description of a language:  

- $T$: set of tokens or **terminals**
- $N$: set of **nonterminals**, symbols representing “sub-languages”
- $P$: set of **productions**, rules for producing strings
- $S$: **start symbol**, a nonterminal that denotes the entire language ($S \in N$)

### Kinds of Grammars

There are 4 major kinds of grammars, depending upon the form of the productions:

- Let $\alpha, \beta, \gamma$ denote strings of grammar symbols (from $N$ and/or $T$)
- Let $A, B$ denote a single nonterminal
- Let $a, b$ denote a single terminal

<table>
<thead>
<tr>
<th>if all the productions have the form</th>
<th>the grammar is called</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \to a$ or $A \to aB$</td>
<td>regular</td>
<td>3</td>
</tr>
<tr>
<td>$A \to \alpha$</td>
<td>context-free</td>
<td>2</td>
</tr>
<tr>
<td>$\alpha \beta \to \alpha \gamma \beta$</td>
<td>context-sensitive</td>
<td>1</td>
</tr>
<tr>
<td>(no restriction)</td>
<td>general</td>
<td>0</td>
</tr>
</tbody>
</table>

Each type of grammars includes all grammars of larger type #

- Regular grammars describe the same set of strings as do regular expressions.
- General parsing techniques exist for regular and context-free languages.
  - Lexemes usually described by regular expressions
  - Modern HLL’s described via context-free grammars

### Backus-Naur Form

**Used to describe the syntax of ALGOL 60.**

- Nonterminals are enclosed in ( )
  - e.g., ⟨expression⟩, ⟨statement⟩
- Terminals are written “as is” or quoted e.g., +, -, ‘<’
- The “can be” symbol is ::=.
- The notation
  $$\langle \text{nonterm} \rangle ::= \text{string}_1|\text{string}_2|...$$

is understood as shorthand for

$$\langle \text{nonterm} \rangle ::= \text{string}_1$$
$$\langle \text{nonterm} \rangle ::= \text{string}_2$$
$$\langle \text{nonterm} \rangle ::= ...$$

### Sample Productions

$$\langle \text{statement} \rangle ::= \langle \text{assignment} \rangle | \langle \text{if-stmt} \rangle | \langle \text{loop-stmt} \rangle$$

$$\langle \text{assignment} \rangle ::= \langle \text{expr} \rangle ::= \langle \text{expr} \rangle ;$$
$$\langle \text{loop-stmt} \rangle ::= \text{while}( \langle \text{expr} \rangle ) \langle \text{statement} \rangle$$

### Writing Grammars

Not as mysterious as it may seem:

- Nonterminals name “meaningful” substrings
- The structure of the grammar should intuitively reflect the structure of the sentences

**Example:** Consider part of a telephone directory:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smilling, Joe T</td>
<td>207 Elm St.</td>
<td>555-1201</td>
</tr>
<tr>
<td>Smit, Robert</td>
<td>12 Geneva Ave.</td>
<td>555-2345</td>
</tr>
<tr>
<td>Sallie</td>
<td>143 Whit Landing Rd.</td>
<td>555-7834</td>
</tr>
<tr>
<td>Smith, Andrew A</td>
<td>427 1st St.</td>
<td>555-8928</td>
</tr>
<tr>
<td>Arthur B</td>
<td>123 Sesame St.</td>
<td>555-1234</td>
</tr>
<tr>
<td>Barbara K</td>
<td>476 Rock Lake Dr.</td>
<td>555-4829</td>
</tr>
</tbody>
</table>

What can we say about the structure of this sentence?

We are defining the language of “TelephoneDirectories”:

$$\langle \text{phoneDir} \rangle ::= \ldots$$
We can recognize certain internal structures: names, addresses, phone numbers, spacers.

It may be tempting to expand things all the way down to the character level:

But that's neither necessary nor desirable.

- Need to ask: What are the tokens in this language?

We can recognize certain internal structures: names, addresses, phone numbers, spacers.

Next, we note that there are really 2 different kinds of lines here:

And these 2 kinds of lines are arranged into a definite pattern:

And finally, we note that a phone listing consist of repeated such blocks:
Pulling it all together:

\[
\langle \text{phoneDir} \rangle ::= \langle \text{blockList} \rangle
\]

\[
\langle \text{blockList} \rangle ::= \langle \text{nameBlock} \rangle
\quad | \quad \langle \text{nameBlock} \rangle \langle \text{blockList} \rangle
\]

\[
\langle \text{nameBlock} \rangle ::= \langle \text{fullLine} \rangle \langle \text{partialLineList} \rangle
\]

\[
\langle \text{partialLineList} \rangle ::= \langle \text{partialLine} \rangle \langle \text{partialLineList} \rangle
\quad | \quad \langle \text{partialLine} \rangle
\]

\[
\langle \text{fullLine} \rangle ::= \langle \text{name} \rangle \langle \text{address} \rangle
\quad \text{spacer phoneNum}
\]

\[
\langle \text{partialLine} \rangle ::= \langle \text{partialName} \rangle \langle \text{address} \rangle
\quad \text{spacer phoneNum}
\]

\[
\langle \text{partialName} \rangle ::= \langle 1\text{stNm} \rangle \langle \text{mi} \rangle
\quad | \quad \langle \text{lastNm} \rangle \langle 1\text{stNm} \rangle \langle \text{mi} \rangle
\]

\[
\langle \text{name} \rangle ::= \langle \text{lastNm} \rangle , \langle 1\text{stNm} \rangle \langle \text{mi} \rangle
\quad \text{noBlankString}
\]

\[
\langle \text{address} \rangle ::= \langle \text{stNumber} \rangle \langle \text{street} \rangle
\quad \text{noBlankString}
\]

\[
\langle \text{stNumber} \rangle ::= \text{char}
\quad \text{noBlankString}
\]

\[
\langle \text{street} \rangle ::= \text{string}
\quad \text{string}
\]

How does this compare to our earlier definition of a grammar = (N,T,S,P)?

- N is the set of nonterminals: \langle \text{phoneDir} \rangle, \langle \text{blockList} \rangle, ..., \langle \text{street} \rangle
- T is the set of terminals (tokens): \text{spacer}, \text{phoneNum}, \text{string}, ...
- S is the starting nonterminal: \langle \text{phoneDir} \rangle
- P is the set of productions we have just written

Using Grammars
How do we know when a grammar describes what we want?

- Need to show that the grammar generates the strings in our language.
  - begin with start symbol
  - expand nonterminals using production rules
  - continue until all nonterminals have been removed

Recursion in CFG’s
Note that CFG’s are often recursive:

\[
\langle \text{stmt} \rangle ::= \langle \text{loop-stmt} \rangle
\quad \text{while} \left( \langle \text{expr} \rangle \right) \langle \text{stmt} \rangle
\]

\[
\quad \text{...}
\]

\[
\quad \text{...} \quad \text{while} \left( \langle \text{a} \rangle > 0 \right) \langle \text{stmt} \rangle
\]

\[
\quad \text{...} \quad \text{while} \left( \langle \text{a} \rangle > 0 \right) \langle \text{assignment} \rangle
\]

\[
\quad \text{...} \quad \text{while} \left( \langle \text{a} \rangle > 0 \right) \langle \text{expr} \rangle := \langle \text{expr} \rangle ;
\]

Recursion in a grammar may be

- essential, because self-inclusion is part of the abstract syntax
- incidental, because recursion is used to capture repetition
Note how recursion is used here, where our intuitive idea is “repetition”:

\[
\langle \text{stmt} \rangle ::= \{ \langle \text{stmt-seq} \rangle \}
\]

\[
\langle \text{stmt-seq} \rangle ::= \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle \langle \text{stmt-seq} \rangle
\]

This does indeed generate the strings we want, but it fails to reflect the structure we want.

This would be a parse tree for \( a + 2 \ast b \):

Ambiguity

A grammar is ambiguous if it allows more than one parse tree for some string in its language.

The simple expression grammar is ambiguous because it fails to reflect the rules of associativity and precedence that we use to interpret infix expressions.

**Associativity in Grammars**

Start with a simpler case: expressions involving \(-\) only.

Compare parse trees for \( a - b - c \) using the 2 grammars:

\[
\langle \text{exp} \rangle ::= \langle \text{term} \rangle - \langle \text{exp} \rangle \\
\langle \text{exp} \rangle ::= \langle \text{exp} \rangle - \langle \text{term} \rangle \\
\langle \text{term} \rangle ::= \text{id} \\
\langle \text{term} \rangle ::= \langle \text{term} \rangle \langle \text{term} \rangle
\]

**Gratmars for Expressions**

The simplest approach to representing expressions would be

\[
\langle \text{exp} \rangle ::= \text{id} | \text{number} \\
\mid \langle \text{exp} \rangle \langle \text{op} \rangle \langle \text{exp} \rangle \\
\mid (\langle \text{exp} \rangle) \\
\langle \text{op} \rangle ::= + | - | \ast | / \\
\]

But this is also a parse tree for \( a + 2 \ast b \):

\[
\langle \text{exp} \rangle ::= \langle \text{term} \rangle \langle \text{term} \rangle \\
\langle \text{term} \rangle ::= \text{id} \\
\langle \text{term} \rangle ::= \langle \text{term} \rangle \langle \text{term} \rangle
\]

Compare this

\[
\langle \text{stmt-seq} \rangle ::= \langle \text{stmt} \rangle \langle \text{stmt-seq} \rangle
\]

to

\[
\langle \text{stmt-seq} \rangle ::= \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle ; \langle \text{stmt-seq} \rangle
\]

The right hand side of a production can be empty!

Compare this

\[
\langle \text{stmt-seq} \rangle ::= \langle \text{stmt} \rangle \langle \text{stmt-seq} \rangle
\]

to

\[
\langle \text{stmt-seq} \rangle ::= \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle ; \langle \text{stmt-seq} \rangle
\]
Basic Components — Syntax

Precedence in Grammars
Consider expressions involving + and * only.
The grammar
\[
\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \\
| \langle \text{term} \rangle \\
\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \\
| \langle \text{factor} \rangle \\
\langle \text{factor} \rangle ::= \text{id} \\
\]
uses different “levels” of recursion to group * more tightly than +.

We’re already stuck, because there’s no way that either \( \langle \text{term} \rangle \) or \( \langle \text{factor} \rangle \) will ever expand to a string containing a “+”.

A Full Expression Grammar
\[
\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \\
| \langle \text{exp} \rangle - \langle \text{term} \rangle \\
| \langle \text{term} \rangle \\
\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \\
| \langle \text{factor} \rangle \\
\langle \text{factor} \rangle ::= \text{id} \text{number} \\
| ( \langle \text{exp} \rangle ) \\
\]

Note that, with this grammar, we cannot get a wrong interpretation of \( a + b \ast c \), such as \((a + b) \ast c\):
Basic Components — Syntax

Extended BNF (EBNF)
Some convenient extensions to BNF form:

- {...} represents 0 or more repetitions
- [...] represents an optional part (0 or 1 occurrence)
- (...) used for grouping

Using EBNF, we can rewrite the grammar for statement sequences:

\[
\langle \text{stmt} \rangle \ ::= \ \{ \langle \text{stmt-seq} \rangle \} \\
\langle \text{stmt-seq} \rangle \ ::= \ \langle \text{stmt} \rangle \\
\quad \quad \quad | \ \langle \text{stmt} \rangle \langle \text{stmt-seq} \rangle
\]
as

\[
\langle \text{stmt} \rangle \ ::= \ '{' \{ \langle \text{stmt-seq} \rangle \} '{'}
\]

Repetition with separators is only slightly more complicated in EBNF:

Parameter list for a function call:

\[
\langle \text{params} \rangle \ ::= \ ( \langle \text{paramlist} \rangle ) \\
\langle \text{paramlist} \rangle \ ::= \ \langle \text{exp} \rangle \\
\quad \quad \quad | \ \langle \text{exp} \rangle , ( \langle \text{paramlist} \rangle )
\]
in EBNF becomes

\[
\langle \text{params} \rangle \ ::= \ ( \langle \text{exp} \rangle \{ , \langle \text{exp} \rangle \})
\]

Of course, in most languages, we can supply an empty list () of function parameters:

\[
\langle \text{params} \rangle \ ::= \ (\langle \text{paramlist} \rangle ) \\
\langle \text{paramlist} \rangle \ ::= \ | \ \langle \text{paramlist} \rangle \\
\langle \text{paramlist} \rangle \ ::= \ \langle \text{exp} \rangle \\
\quad \quad \quad | \ \langle \text{exp} \rangle , \langle \text{paramlist} \rangle
\]
in EBNF becomes

\[
\langle \text{params} \rangle \ ::= \ ([\langle \text{exp} \rangle , \langle \text{exp} \rangle])
\]

2.2 Recursive Descent Parsing
A simple parsing technique for (some) context-free grammars:

- For each nonterminal \( \langle N \rangle \), we write a function
  \[
  \text{bool } N(\text{Scanner} \& \text{scanner});
  \]
to recognize if the next sequence of tokens belongs to that non-terminal set.
- The body of \( N(\ldots) \) is derived from the productions with \( \langle N \rangle \) on the left.

Assume we have an appropriate token class...

```cpp
struct Token {
  enum Kinds { integer, string, plus, ... };  
  // depends on the language  
  Kinds kind;  
  string lexeme;  
};
```

... and a scanner:

```cpp
class Scanner {
public:
  Scanner(istream &);  
  Token peek (int numTokens) const;  
  bool match (Token::Kinds kind);  
private:
  :  
};
```

- peek (k) shows us the \( k \)th next token.
– Often, scanners can only “look ahead” one token

• match(t) checks to see if peek(1).kind == t.

– If so, the next token is discarded and match returns true.

– If not, match returns false and the scanner state is unchanged.

In recursive descent parsing, we write recognition functions to match each nonterminal against strings of tokens, and use Scanner::match to recognize a terminal.

A production like

\[ \langle N \rangle ::= \langle S \rangle \ T \langle U \rangle \]

is handled like this:

```cpp
bool N(Scanner& scanner) {
    ...
    return S(scanner) && scanner.match(Token::T) && U(scanner);
    ...
}
```

The tricky part is figuring out which production to use if there’s more than 1 for \( \langle N \rangle \).

For the expression grammar:

\[
\begin{align*}
\langle \text{exp} \rangle & ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \\
& \quad \mid \langle \text{exp} \rangle - \langle \text{term} \rangle \\
& \quad \mid \langle \text{term} \rangle \\
\langle \text{term} \rangle & ::= \langle \text{term} \rangle \ * \langle \text{factor} \rangle \\
& \quad \mid \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & ::= \langle \text{item} \rangle \ * \langle \text{factor} \rangle \\
& \quad \mid \langle \text{item} \rangle \\
\langle \text{item} \rangle & ::= \text{id} | \text{number} \\
& \quad \mid (\langle \text{exp} \rangle)
\end{align*}
\]

...we would have functions:

```cpp
bool exp(Scanner& scanner);
bool term(Scanner& scanner);
bool factor(Scanner& scanner);
bool item(Scanner& scanner);
```

Let’s look at \( \langle \text{item} \rangle \):

\[
\langle \text{item} \rangle ::= \text{id} | \text{number} \\
\quad \mid (\langle \text{exp} \rangle)
\]
Basic Components — Syntax

\[ \langle \text{phoneDir} \rangle ::= \langle \text{blockList} \rangle \]

```cpp
bool phoneDir ( Scanner& scanner )
{
    return blocklist ( scanner );
}
```

\[ \langle \text{blockList} \rangle ::= \langle \text{nameBlock} \rangle |
\langle \text{nameBlock} \rangle \langle \text{blockList} \rangle \]

```cpp
bool blocklist ( Scanner& scanner )
{
    if ( nameBlock ( scanner ) )
    {
        if ( scanner . peek ( 1 ) )
            return blocklist ( scanner );
        else
            return true;
    } else
    return false;
}
```

\[ \langle \text{nameBlock} \rangle ::= \langle \text{fullLine} \rangle \langle \text{partialLineList} \rangle \]

```cpp
bool nameBlock ( Scanner& scanner )
{
    return fullLine ( scanner )
    && partialLineList ( scanner );
}
```

\[ \langle \text{partialLineList} \rangle ::= |
\langle \text{partialLine} \rangle \langle \text{partialLineList} \rangle \]

```cpp
bool partialLineList ( Scanner& scanner )
{
    if ( scanner . peek ( 2 ) == Token :: comma )
        return true;
    else
    return partialLine ( scanner )
    && partialLineList ( scanner );
}
```

\[ \langle \text{fullLine} \rangle ::= \langle \text{name} \rangle \langle \text{address} \rangle \]

```cpp
bool fullLine ( Scanner& scanner )
{
    return name ( scanner ) &&
    address ( scanner ) &&
    scanner . match ( Token :: spacer ) &&
    scanner . match ( Token :: phoneNum );
}
```

2.3 Miscellaneous notes

- Text uses if-then-else to explore ambiguity
- Syntax charts
- Many editors and Unix commands use RE’s for text manipulation.
- Programs exist to translate RE’s and CFG’s into
  - compiler code
  - test data