Syntax

Programming Language Design 2022

Hans Hüttel
incorporating material by Fritz Henglein and Torben Æ. Mogensen

February 14 2022

Learning goals

- To be able to explain the notions of lexical and grammatical syntax and how they differ
- To be able to apply the definitions of lexical and grammatical syntax to explain the central aspects of the syntax of a program in a given programming language
- To be able to use and apply the terminology in programming language syntax
- To be able to explain the general structure of syntax analysis and its phases
- To be able of assess the possible choices in the design of the syntax of a programming language

Part I

Syntax – What is it?

A tiny Haskell program that has *everything*

```
pythtriple a b c = (sq a + sq b == sq c)
                   where
                     sq x = x * x
```

What do we mean by syntax?

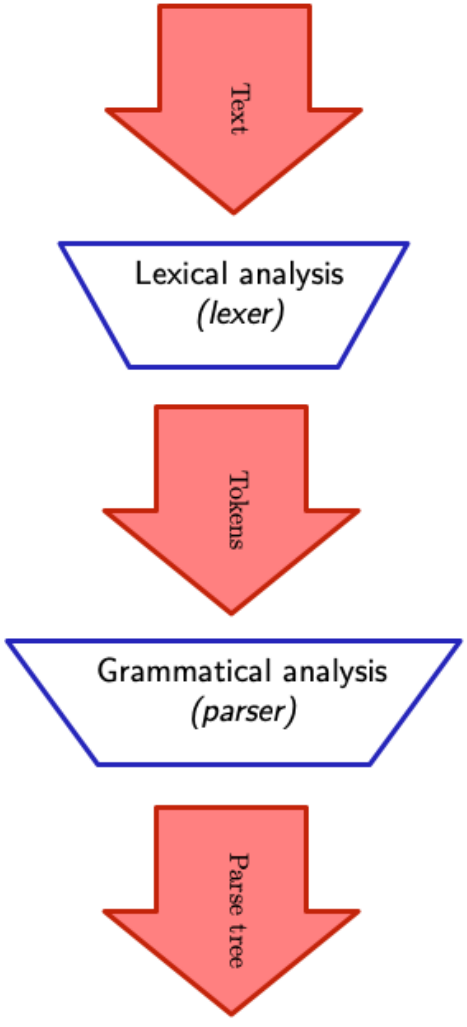
> *The arrangement of words and phrases to create well-formed sentences in a language.*

The word comes via French or late Latin from Greek σύνταξις , from σύν 'together' and ταξις 'arrangement'.

So in the study of syntax we are concerned with how sentences are formed but not with their meaning.

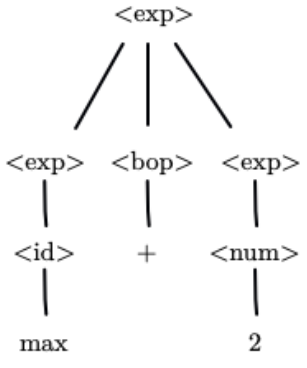
In the world of programming languages, we do not have sentences but instead consider e.g. expressions, statements, declarations and programs.

Syntax analysis



max+2

max + 2



Syntax: Usability vs. implementation

We would like the syntax of our programming to be

Usable It should be easy for us to express algorithms and data using the syntax of our language.

Implementable It should be easy to carry out syntax analysis.

This is a consideration that is at the heart of programming language design. It must hold at both the lexical and the grammatical level.

Part II

The lexical level

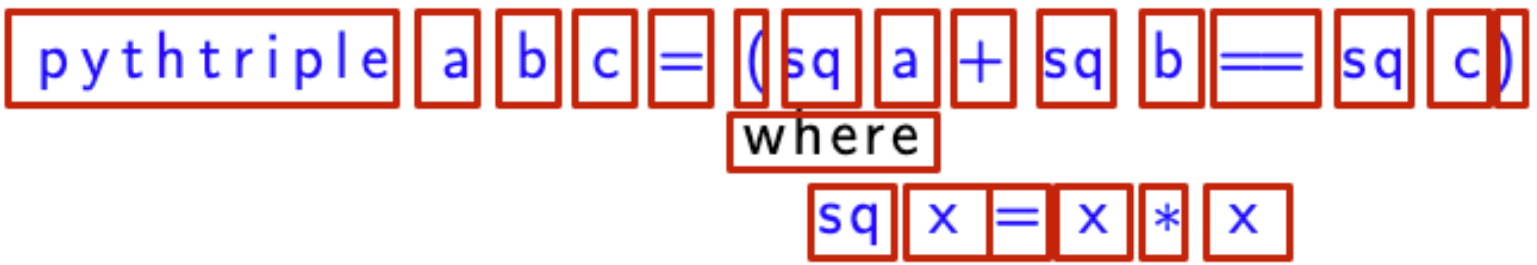
Syntax at the level of individual characters

We can think of a program as a string of text – a sequence of characters.

```
pythtriple a b c = (sq a + sq b == sq c)
                   where
                     sq x = x * x
```

Syntax at the lexical level

But characters can be composed to form *tokens* – these can be built from multiple characters.



Syntax at the lexical level

At the lexical level we consider how to define and analyze a string of characters as a sequence of tokens.

We are also interested in how characters are classified (e.g. as letters or as digits).

Comments are usually not considered as tokens. They are scanned and then discarded during lexical analysis.

Whitespace separates tokens but is not a token in itself.

Defining tokens

The set of valid tokens is defined using *regular expressions*.

There are many different versions of this notation. But they all use at least the operators Kleene star, union and concatenation.

Here is an example – a regular expression describing the shape of identifiers in some programming language:

$$(a \cup b)(a \cup b)^*(0 \cup 1)^*$$

Recognizing tokens

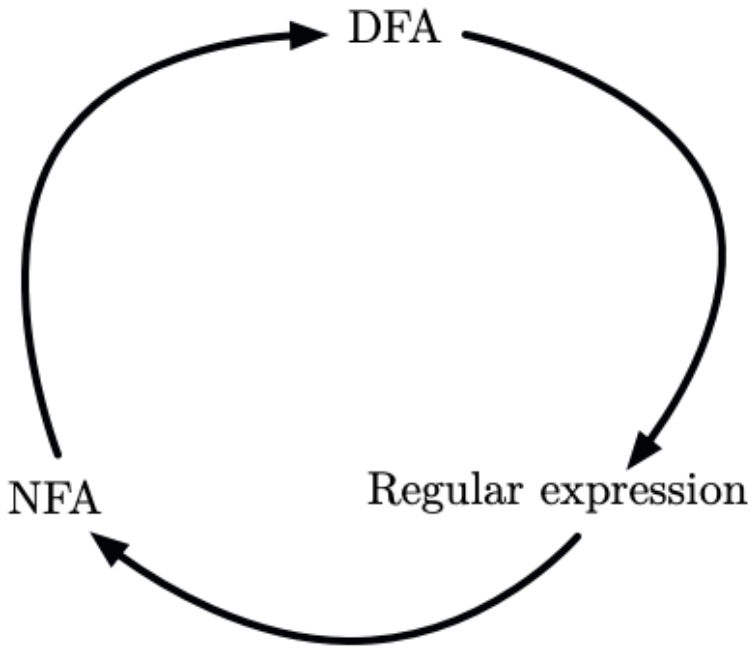
The part of syntax analysis that takes a string of characters and attempts to discover the sequence of tokens that it corresponds to is lexical analysis.

Lexical analysis uses a deterministic finite-state automaton (DFA).

Recognizing tokens

Kleene's theorem from formal language theory tells us that there is an algorithm that, given a regular expression R, will build a DFA M_R that recognizes precisely the set of tokens defined by R.

The theorem involves the notion of a nondeterministic automaton NFA – an intermediate step in building a DFA. (We do not give the details here.)



Separating tokens

How do you know when a token is complete and a new begins?

Is where two tokens, w and here, or the token where?

Separating tokens

In most languages, a token is the *longest prefix* that matches a regular expression (in a set of regular expressions).

The longest-prefix rule can be a problem when operators are not separated by spaces, such as in x+=++*++y.

A solution is to view the entire sequence of operator characters as single operator, and report an error if this not defined.

Another solution is to require that every operator must be a single character.

Older languages

Things were different in the past.

- In APL every token must be a single character (APL, except names and numbers)
- FORTRAN ignores whitespace but recognises keywords that allow a line to be parsed. `DO20I` is split into `DO 20 I` if the line can then be parsed as a do-loop, otherwise it is a name.
- BASIC extracts keywords first. `FORMATION` becomes `FOR MATION`.

Part III

Other lexical concerns

Character sets

The same programming language may specify several different character sets / typographical variants to be used in different contexts (e.g, program editing and program presentation).

There are several considerations when choosing a character set.

- Ease of entry using standard keyboards and similar devices
- Ease of display/print on standard devices
- Richness of character set
- Multi-character symbols
- Typographical variants of symbols (font, colour, size, subscripted, . . .)

CASE sensitivity

In English, Danish, German and other natural languages, UPPER and lower case letters usually have the same meaning, but there are rules for when to capitalise, and sometimes capitalisation can affect meaning.

What about programming languages?

CASE sensitivity

In English, Danish, German and other natural languages, UPPER and lower case letters usually have the same meaning, but there are rules for when to capitalise, and sometimes capitalisation can affect meaning.

What about programming languages?

- Some languages are case-insensitive, i.e., `BEGIN` and `begin` have the same meaning.
- In some languages, case matters, and keywords have a mandated case, but you are free to write identifiers using Mixed-Case.

CASE sensitivity

In some languages, capitalisation is used to distinguish different kinds of identifier (such as variable, type, or constructor).

In Haskell, type constructors and term constructors are capitalised. Other functions are not.

```
data Tree = Leaf Integer | Branch Tree Tree

tree (Leaf x) = x
tree (Branch t1 t2) = (tree t1) + (tree t2)
```

Identifiers

Identifiers are user-defined names – names of variables, functions, etc.

Usually identifiers are constructed as a letter followed by any number of letters and digits.

Some variations are

- Additional symbols (such as spaces, hyphens, underscores, quotes, ?, . . .) are often allowed in names.
- Other languages (TEX, Troll, . . .) allow only letters in names.
- Operator names can be built from non-alphanumeric characters.

Coding styles can add additional recommendations on form.

Whitespace

In most languages, whitespace separates tokens, but is otherwise insignificant (except in string constants). But there are variations.

Older languages sometimes have special conventions.

- FORTRAN ignores spaces.
- ALGOL 68 allows spaces in variable names.
- End-of-line may terminate statements (BASIC, FORTRAN, ...).

Comments

Comments/remarks usually have no semantic meaning, but carry information for human readers. Comments can have several forms:

- Line comments
- Bracketed comments
- Comments that carry information for compilers or other tools
- Literate programming

One needs to think about how nested comments should be handled! Do we even want to have them?

```
/*
                outer comment
                /* inner comment */
                is outer comment continued?
*/
```

Reserved symbols

In most programming languages, certain sequences of characters are special tokens – they are reserved symbols. These are keywords (if , **while**, . . .) brackets and punctuation (such as semicolon).

Some languages (such as PL/I) allow keywords to be redefined.

This has both advantages and disadvantages:

For	If there are many reserved words, it may be easy to use one as a variable name without meaning to do so, so allowing redefinition gives fewer surprises.
For	If new keywords are added to the language, this won't break old programs.
Against	Redefining a keyword to be a variable can make a program hard to read.
Against	It can give hard-to-detect errors!

Re-using keywords as identifiers

A possible solution is to have identifiers and keywords built from disjoint character sets. ALGOL 68, for example, had CAPITALISED, 'quoted' or **boldface** keywords, but lower-case variables.

This way you could define a variable called **while**, if you really, really want to. It would never be confused with a **WHILE** loop.

Indentation

Some languages have indentation-sensitive syntax: Haskell, F#, Python, ...

```
pythtriple a b c = (sq a + sq b == sq c)
                   where
                     sq x = x * x
```

Indentation

Some languages have indentation-sensitive syntax: Haskell, F#, Python, ...

```
pythtriple a b c = (sq a + sq b == sq c)
                   where
                     sq x = x * x
```

This is not just whitespace! It is a form of *implicit bracketing.* Using it instead of explicit bracketing can reduce the number of lines of code and give a more readable layout.

But it can make error messages less understandable.

Haskell also allows for implicit bracketing.

The lexical level has its limits!

Not all aspects can be handled at the lexical level.

If we want to ensure that parentheses are balanced, such that the Lisp expression

```
(a (b c) (c de) f)
```

is allowed, but the Lisp expression

```
sunn o)))
```

is not, we run into problems. Regular expressions cannot express this. This is a consequence of the Pumping Lemma for regular languages from formal language theory.

We have to deal with such issues at the grammatical level.

Part IV

The grammatical level

Grammatical elements

At the grammatical level we describe how tokens are combined to form a structure.

Different types are:

- Line-based syntax (FORTRAN, BASIC)
- Multi-line syntax (Plankalkül)
- Imitating natural language (COBOL)
- Bracketed syntax (LISP, HTML)
- Prefix, postfix, and operator-precedence syntax (LOGO, Forth, SNOBOL)
- Context-free syntax (ALGOL 60 and onwards)
- Visual languages (Scratch)

Line-based syntax

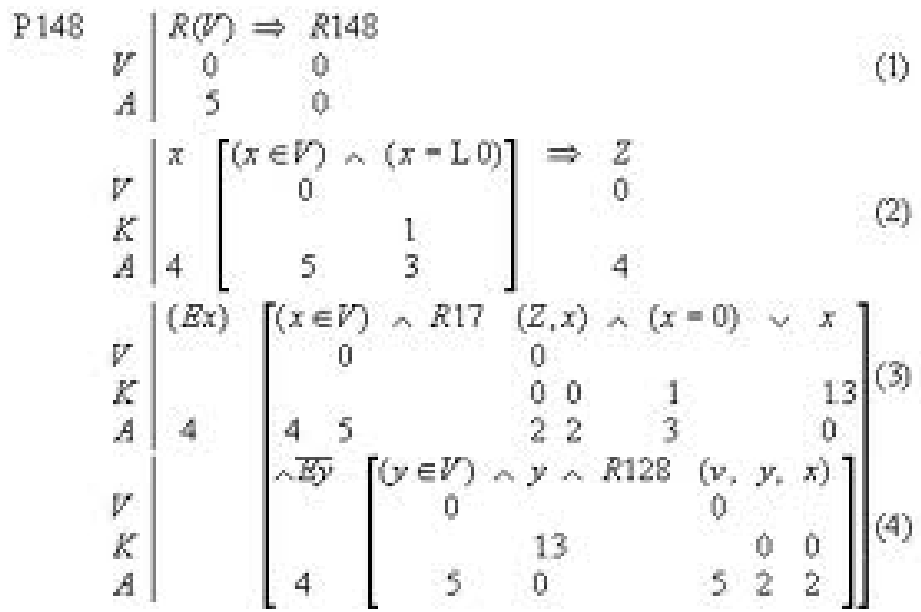
Line-based syntax has one statement per line. This an old approach used in e.g, FORTRAN or BASIC:

C ← FOR COMMENT / STATEMENT NUMBER (1–5)	CONTINUATION (6)	FORTRAN STATEMENT (7–72)	IDENTIFICATION (73–80)
C		PROGRAM FOR FINDING THE LARGEST VALUE	
C	X	ATTAINED BY A SET OF NUMBERS	
		BIGA = A(1)	
		DO 20 I = 2,N	
		IF (BIGA - A(I)) 10, 20, 20	
10		BIGA = A(I)	
20		CONTINUE	

Some constructs (such as loops) can span multiple lines, but, in effect, the beginning and end of these constructs are treated as separate statements (like FOR-NEXT in BASIC).

Multi-line syntax

This is like line-based syntax, but a statement consists of multiple lines, each with its own meaning. Related elements line up vertically.



We only know of Konrad Zuse's Plankalkül in this category.

Syntax that imitates natural language

The motivation to make programs readable (and possibly programmable) by non-programmers. Examples: COBOL, AppleScript, HyperTalk, Inform 7.

COBOL

Here is the "Hello, world" program in Cobol.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. HELLO.
 PROCEDURE DIVISION.
       DISPLAY "Hello, world".
 END PROGRAM HELLO.
```

Syntax that imitates natural language

But . . .

- The meaning of a program will often differ subtly from that of the same natural language statement.
- We cannot write arbitrary English.
- This gets verbose!
- And there is much more to programming than syntax.

Bracketed syntax

Every subexpression or statement is explicitly enclosed in brackets.
Examples: LISP, Scheme, Clojure, HTML, XML, LaTeX.

Lisp dialects such as Scheme use bracketed syntax exclusively.

Here is the factorial function in Scheme.

```
(define factorial
   (lambda (n)
      (if (= n 0) 1
                  (* n (factorial (- n 1))))))
```

Bracketed syntax

This, too, has its pros and cons.

For	Easy to parse (also for humans).
For	Syntax is easy to manipulate by programs.
Against	Verbose.
Against	Does not exploit human experience with natural languages and mathematical notation.

Prefix, postfix and infix operator syntax

Arithmetic expressions where operators have fixed arity (number of arguments) can be written unambiguously using prefix or postfix notation without brackets:

Infix notation: $2 * (3 + 4!) - 5$

Prefix notation: $- * 2 + 3\ !\ 4\ 5$

Postfix notation: $2\ 3\ 4\ !\ + * 5\ -$

Prefix, postfix and infix operator syntax

Prefix and postfix notation are compact and easy to parse by machine. Example languages:

Logo (prefix): `setxy sum :x quotient :y 2 :z`

PostScript (postfix): `0 0 moveto 0 40 mm lineto stroke`

But is this human-friendly?

Prefix, postfix and infix operator syntax

Prefix and postfix notation are compact and easy to parse by machine. Example languages:

Logo (prefix): `setxy sum :x quotient :y 2 :z`

PostScript (postfix): `0 0 moveto 0 40 mm lineto stroke`

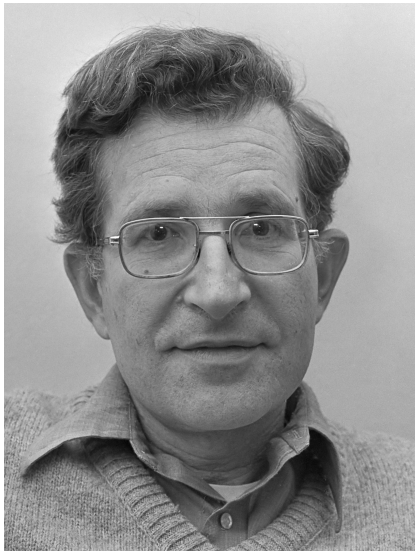
But is this human-friendly?

An alternative is to use unary and binary (infix) operators with precedence and associativity (overridden by brackets) for all syntax.

SNOBOL is an old language that uses this compromise.

Context-free syntax

Context-free grammars arose in the work of Noam Chomsky in the 1950s. He wanted to understand the syntax of natural languages.



The creators of ALGOL60 – Peter Naur, John Backus and others – used Backus-Naur Form (BNF) to define the syntax of their language. Backus knew of Chomsky's work.

Context-free syntax

A context-free grammar specifies a collection of nonterminals that correspond to syntactic categories and production rules that tell us how elements of these categories must be built.

Here is a screenshot from the ALGOL60 report.

```
<procedure identifier> ::= <identifier>
<actual parameter> ::= <string>|<expression>|<array identifier>|
        <switch identifier>|<procedure identifier>
<letter string> ::= <letter>|<letter string><letter>
<parameter delimiter> ::= ,  |  )<letter string> :(
<actual parameter list> ::= <actual parameter>|
          <actual parameter list><parameter delimiter><actual parameter>
<actual parameter part> ::= <empty>|(<actual parameter list>)
<function designator> ::= <procedure identifier><actual parameter part>
```

Context-free syntax – Pros and cons

For Grammars describe complex rules for forming syntax.

For Parsers can be built automatically from grammars by a parser generator.

Against A grammar can be ambiguous. It is algorithmically undecidable if a grammar is ambiguous! Parser generators only work with well-behaved grammar formats that are known not to allow ambiguities.

Against Parsing may require cubic time complexity ($O(n^3)$), so we usually consider *deterministic context free grammars*. Here, parsing can be done in time $O(n)$.

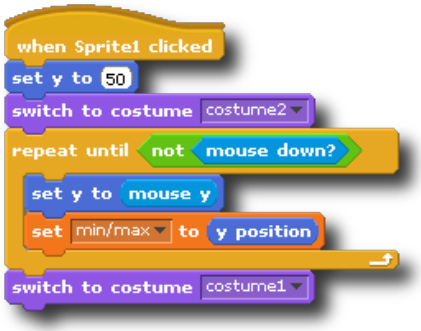
? Lookahead.

Most modern languages are defined using context-free grammars, but may use additional disambiguation rules such as operator precedence or classification of identifiers (type / variable / ...).

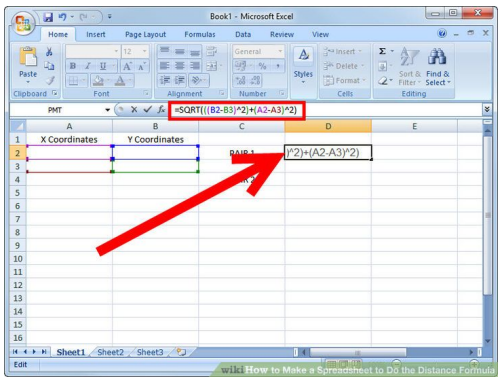
Visual languages

Visual programming languages do not use (pure) text to express syntax, but uses graphical elements.

Scratch:



Spreadsheets:



Such languages require special editors.

Part V

Other concerns at the grammatical level

Bracketing symbols

When statements/expressions are explicitly delimited, there are several possible choices of bracketing:

Uniform bracketing:	Use **begin**/**end**, (/) or {/} for all statements/expressions.
Statement-specific terminators:	**if**/**fi**, **if**/**endif**, / , ...
Indentation:	Bracketing by increase/decrease of indentation.

Or a mix thereof.

Operator precedence

Operator precedence and associativity reduce the need for explicit bracketing, but it can be hard for a human to remember a large number of precedence rules.

Language	# of operators	# of levels
C++	> 50	16
Haskell	extendable	10
Pascal	17	4
Smalltalk	extendable	1

Operator precedence

How are user-defined operators given precedence?

C++	No new operator names, but may overload predefined operators.
Haskell	Explicitly declared precedence and associativity.
Smalltalk	All the same
F#	Depends on initial character (mostly)

Macros

Macros can concern both lexical and grammatical structure. In C, macros are expanded before lexical analysis, so they can lead to unexpected behaviour.

```
#define square(X)X*X
```

will give the following expansions:

square(f())	expands to	f()*f()
square(x+y)	expands to	x+y*x+y
square(/)	expands to	/*/

Additionally, macros are dynamically scoped (we return to scope rules in a later session). *Hygienic macros* (Scheme, Rust) address these issues.

Many languages do perfectly well without macro systems.

Beyond context-free syntax

We saw that regular expressions cannot capture all aspects of programming language. The same is the case for context-free grammars.

A non-context-free feature of most programming languages is that of declaration before use.

We should be able to rule out programs such as

```
int x;
y = 7
```

in which the variable y is used, but never declared.

Unfortunately, no context-free grammar can capture this property. This follows from the Pumping Lemma for context-free languages, which is a theorem in formal language theory.

Beyond context-free syntax

When we want to specify and check for syntactic properties of this kind, we need something more powerful than context-free grammars.

There are so-called context-sensitive grammars which extend context-free grammars, but general parsing algorithms for these have very high time complexity.

A better, and widely alternative is to use so-called attribute grammars. Many parser generators support various forms of attribute grammars.

The good, the bad and the not so sensible

Some pitfalls in the design of syntax:

- Having similar but subtly different syntactic constructs (such as `fun` and `function` in F#).
- Parsing depends on the declaration of a symbol that may be far away. C++ example: x = a<b,c>(d);
- Special cases. For example, in F#, the infix constructor symbol :: looks like an infix operator, but you can not use it as such in all contexts.
  For example, you can write map (+) ll but not map ((::) ll, and it is the only infix constructor symbol.
- Arbitrary limits, such as length of names, number of nested brackets, number of parameters, ...
- Extreme brevity or verbosity.