Syntax

Programming Language Design 2021

Fritz Henglein
Slides based on Torben Mogensen

February 15th, 2021

What is syntax?

- **(Generative) syntax**: Rules for (explaining/analyzing) the arrangement of words and phrases to create well-formed sentences in a language.
- What are words, phrases, sentences, rules in programming languages?
- Customary concepts used in programming languages:
  - Character set: What are the atomic symbols (characters) of the language?
  - Tokens: How are characters combined to form classes (token classes) of well-formed words (tokens)?
  - Whitespace: How are tokens separated from each other (whitespace)?
  - Comments: How is text provided that has no bearing on the program semantics (comments, documentation)?
  - Grammar: How are well-formed phrases and sentences specified (grammar)?

Lexical and grammatical aspects of syntax

Lexical structure: Character set, tokens and whitespace, to form multi-character token classes such as names, numbers, keywords that carry an indivisible meaning ("one thing"). Usually expressible as a *regular language*.

Grammatical structure: Rules for combining lexical elements to form abstract tree structures for expressions, statements, definitions, modules, whole programs, as a basis for eventually associating semantics to a program (what programs does when executed). Usually expressible as a *context-free language*.

Comments have their own rules so that a program can *conceptually* be processed by:

1. Find and remove all comments.
2. Parse (find lexical and grammatical structure of) remaining program.
3. Compile or interpret program based on its grammatical structure (abstract syntax tree).

Lexical aspects

- Character sets
- Case sensitivity
- Identifiers
- Whitespace
- Reserved symbols (keywords)
- Separation of tokens

Character sets

- Character set: Finite alphabet of atomic symbols, with built-in notion or equality/inequality or total order.
- Examples: ASCII (128 characters); Unicode (143,859 characters), usually with UTF-8 coding.
- Considerations for 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
    - ★ A programming language may specify multiple typographical/visual variants (font, color, size, subscripted, ...) for use in different contexts (such as program editing and program presentation), but usually with a single underlying character set as a finite ordered set in the mathematical sense.

Tokens and token classes

- Token: Well-formed (according to some formation rule) sequence of *visible* characters.
  - Visibility may be broken out of spite/for sheer fun: Whitespace is a programming language whose characters consist exclusively of invisible characters (blank, tab, linefeed; visible characters are ignored.
  - Token classes: Disjoint sets of tokens (each specified as a regular language); e.g.
    - numerals (number constants), identifiers, string constants, bracketing symbols, operators, separators, terminators, etc.

Identifiers

- Identifier: Name that can be bound to something/refers to something (variable, named function, etc.)
- Common formation rule:

$$\mathit{ident} \quad ::= \quad \mathit{start}\ \mathit{letter}^*$$

  where *start* is an alphabetic character and *letter* an alphanumeric character.
- Variations:
  - Additional characters (such as spaces, hyphens, underscores, quotes, `?`, ...) may be permitted in identifiers.
  - There may be limits on the lengths of identifiers (FORTRAN: 6, COBOL: 30).
  - Some languages (TEX, Troll, ...) allow only letters in identifiers.
- Initial letter may be used to distinguish token classes.
  - Haskell: lower case = identifier, upper case = constructor
  - Fortran: `I` – `N` = integer variable, other letters floating point variable.

Coding styles can add additional restrictions on form.

Token equality

- Given tokens from the *same token class* when are they considered equal and when different?
  - Is `BEGIN` always the same as `begin`? Can they always be interchanged without changing the meaning of the program?
  - Are `Blob` and `blob` the same variable or different variables?
- Mathematically: Given the character sequences forming a token class, what is the equivalence relation (on character sequences) that specifies their equality (as tokens)? Examples:
  - Case-sensitivity: Tokens are considered equal only if they are equal as character sequences.
  - Case-insensitivity: Tokens are considered equal if they are equal modulo upper/lower case.
  - Prefix equivalence: Identifiers are considered equal if their first k characters are the same. (Pascal: $k = 8$).
- Note:
  - Keywords may be required to be upper case, but identifiers not. (They are typically different token classes.)
  - Capitalization may be used to distinguish token classes; identifiers versus constructors.

Operators

- Operators: Typically built from operator characters $(=, <, >, +, -, /, *, \ldots)$, but may contain alphanumeric characters.

Whitespace

- Whitespace: Sequence of invisible characters; separate tokens.
- General principle: All nonempty sequences of whitespace characters are equivalent to each other, except in string constants and comments; that is replacing one whitespace sequence by another doesn't change lexical and grammatical structure.
- Exceptions:
  - FORTRAN: All whitespace is equivalent with the empty sequence; that is Fortran ignores (discards) whitespace characters.
  - ALGOL 68: Allows spaces in variable names.
  - BASIC, FORTRAN, ...: End-of-line character is different from other whitespace characters; it may terminate statements
  - COBOL, Haskell, F#, Python, ...: Indentation-sensitive syntax. Number of kind of whitespace characters are significant; e.g. a newline followed by 6 spaces is not necessarily equivalent to having a newline with 7 spaces; a tab may be equivalent to 6 spaces, but not 7 spaces, etc.

Indentation-sensitive syntax

- Using indentation instead of explicit bracketing can reduce the number of lines of code and enforce readable layout, but it can make error messages less understandable and code editing fragile.

Reserved symbols

- Reserved symbol: Token class with few elements.
- Keyword: Reserved word (alphabetic character sequence) that "looks" like an identifier and usually means the same (cannot be rebound).
  - Exception: PL/I, depending on grammatical context (where it occurs) allows a keyword to be used as an identifier.
  - Allowing redefinition has advantages and disadvantages:
    - \+ 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.
    - \+ If new keywords are added to the language, this won't break old programs.
    - − Redefining a keyword to be a variable can make a program hard to read.
    - − It can give hard-to-detect errors.
- Common solution: Identifiers and keywords as separate token classes.
  - ALGOL 68: Has CAPITALIZED, 'quoted' and **boldface** keywords, but lower-case variables.
  - Reserve certain identifiers as keywords (reserved words): (`if`, `while`, . . .), can never be used as identifiers.

Comments

- Comment: Demarkated text (character sequence) with no semantic significance; carries information for human readers and tools processing programs.
- Comments can have several forms:
  - Line comments
  - Bracketed comments
  - Comments that carry information for compilers or other tools
  - Literate programming
- Usually handled by a regular language (processed like a token class equivalent to whitespace).
- But: How are nested comments handled?
  - As a Dyck-language (with pairing off open-close markers)? Not regular.
  - Or as a regular language (without pairing off)?

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

Separation of tokens

- How do you know where a token starts and ends?
- Common solution: The *longest prefix* of the remainder of the program that matches an element of the regular language of tokens (usually described by a regular expression for each token class) is the next token.
- Exceptions:
  - APL: All tokens except identifiers and numerals consist of a single character.
  - FORTRAN: Ignores whitespace, but recognizes keywords that allow a line to be parsed; e.g., `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, so `FORMATION` becomes `FOR MATION`.
- The longest-prefix rule can be problemat when operators are not separated by spaces, such as in `x+=++*++y`. Solutions: Parse the full sequence of operator characters as a single operator, and report error if not defined. Or make all operators single symbols.

Lexical structure: Summary

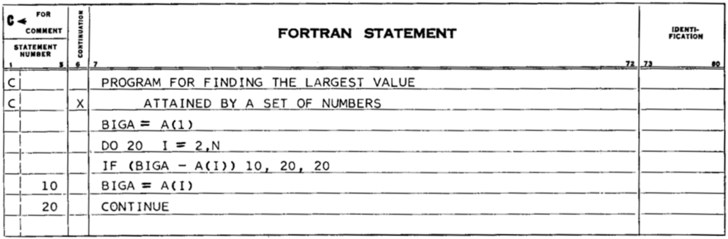
- Token classes: Disjoint regular sets of character sequences with similar function
- Token equality: Equivalence relation on character sequences in same token class
  - Case-sensitive: different character sequences = different tokens
  - Case-insensitive: character sequences only different in upper/lower case = same token
- Whitespace: Invisible character sequences separating tokenn
  - layout insensitive: all whitespace is equivalent
  - layout sensitive: distinction between different whitespace sequences
- Lexical analysis: Splitting input into sequence of tokens

Grammatical structure

- Grammatical structure: Rules for combining tokens into phrases, components and eventually whole programs (statements, expressions, definitions, modules, etc.)
- Well-formedness and formation rules:
  - 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 descendants)
  - Visual languages (Scratch)

Line-Based Syntax

- One statement per line, e.g, FORTRAN or BASIC:




```
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
```

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

Multi-Line Syntax

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

```
P148 | R(V) ⇒ R148
   V |  0      0                                          (1)
   A |  5      0
     | x  [(x∈V) ∧ (x = L 0)]  ⇒  Z
   V |      0                     0
   K |               1                                    (2)
   A | 4 [  5        3       ]    4
     | (Ex)  [(x∈V) ∧ R17 (Z,x) ∧ (x = 0) ∨  x ]
   V |          0          0
   K |                     0 0     1        13 ]          (3)
   A |   4    4 5          2 2     3         0 ]
     |       ∧Ēy [(y∈V) ∧ y ∧ R128 (v, y, x) ]
   V |              0              0
   K |                 13              0  0   ]]          (4)
   A |        4     5  0           5   2  2   ]]
```

- Example: Plankalkül (1945)

Syntax that imitates natural language

- Lexical and grammatical formation rules that generate natural language sentences whose informal natural language semantics is consistent with it formal (programming language) semantics.
- Objective: Readability and use by domain specialists (e.g. business engineers, lawyers, case managers, etc)
- Examples: COBOL, AppleScript, HyperTalk, Inform 7, ACE (Attempto Controlled English).
- Very active resarch area, e.g. pursuit of *Ricardian contracts* with single concrete syntax whose formal semantics is sufficiently consistent with legal reading in practice.
- Historically only partially successful:
  - Meaning may differ subtly from natural-language meaning.
  - May raise expectation that it should accept arbitrary English instead of certain limited English (e.g. ACE).
  - Too verbose for mathematicians, scientists, engineers and computer scientists (if they are the targeted domain specialists).
  - Bad error messages (not given in terms of natural language explanations).

Bracketed Syntax

- Nested constructs (expressions, definitions, etc.) always enclosed by opening/closing pairs of tokens
- Examples: LISP, Scheme, Clojure, HTML, XML, LaTeX.
  - \+ Easy to parse.
    - Applicable to all context-free languages. (Chomsky-Schützenberger Representation Theorem)
  - \+ Syntax is easy to manipulate by recursive functional programs.
  - – Verbose.
  - – Does not exploit human experience with natural languages and mathematical notation.

Example (Scheme):

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

Prefix, postfix and infix operator syntax

- Expressions built from fixed-arity prefix, infix and postfix operators without/with few parentheses.

  Mathematical notation: $2 * (3 + 4!) - 5$
  Prefix notation (Polish notation): $- * 2 + 3 \; ! \; 4 \; 5$
  Postfix notation (Reverse Polish notation): $2 \; 3 \; 4 \; ! + * \; 5 \; -$

- Prefix and postfix notation are compact and easy to parse by a stack machine, but less so by humans.
- Example languages:

  Logo (prefix): `setxy sum :x quotient :y 2 :z`
  PostScript (postfix): `0 0 moveto 0 40 mm lineto stroke`

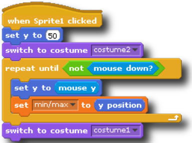
Context-free syntax

- Syntax is specified by a context-free grammar, without open/close tokens.
  - + Can describe complex rules for forming syntax.
  - + Parsers can be built automatically from grammars.
  - – May be ambiguous: Same string may have multiple parses and thus potentially different semantics. (Generally undecidable, but can be approximated or complemented by a choice rule that specifies which among multiple parse must be chosen.)
  - – Context-free parsing is $O(n^3)$ in the worst-case, so grammars often restricted to *deterministic context free grammars* or other finite look-ahead parsing that executes in $O(n)$ time.
- Most modern languages are defined using context-free grammars, combined with disambiguation rules such as operator precedence or classification of identifiers (type/variable/…).

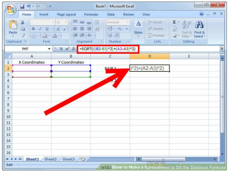
Visual Languages

- Do not use (pure) text to express syntax, but uses graphical elements.
- Enforce grammatical formation rules by visual guides.
- Examples:

Scratch:



Spreadsheets:



- Require special editors.

Other Considerations

- Operator precedence hierarchy.
- Bracketing symbols
- Macros

Operator Precedence Hierarchy

- Operator precedence and associativity reduce the need for explicit parenthesization, 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

- 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)

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**, <p>/</p>, ...

  Indentation: Bracketing by increase/decrease of indentation.
- Or a mix thereof.

Macros

- Macro: Code templates with substitutable parameters.
- Operate at the character (usually) or token sequence level.
- Macro assemblers and language agnostic macro preprocessors operate at the character level: A template can be any character sequence with "holes" for splicing character sequences in.
- In C, macros are expanded before lexical analysis, so it can have unexpected behaviour. Example: `#define square(X) X*X` will give the following expansions:

```
square(f())      f()*f()
square(x+y)      x+y*x+y
square(/)        /*/
```

- Macros are not type checked; code is only type checked after macro expansion.
- *Hygienic macros* (Scheme, Rust): Perform capture avoiding substitution (analogous to lexically scoped functions)
- Many languages do not have a built-in macro system; comparable efficiency is obtained by their compiler performing function inlining.

Syntax design pitfalls: A selection

- Having several similar, but subtly different syntactic constructs (such as `fun` and `function` in F#).
- Parsing depends on declaration of symbol that may be far away. C++ example: `x = a<b,c>(d);`
- Special cases/nonorthogonality. 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 particular length of identifiers, specific number of nested brackets, fixed maximum number of parameters, ...
- Extreme brevity or verbosity, especially when mixed together (e.g. very long identifiers and very shot operators)

Grammatical structure: Summary

- Hierarchical decomposition of whole program into its parts (parsing)
- Context-freeness: Usually describable by context-free grammar (not taking variable bindings and typing into account)
- Various design and implementation approaches:
  - Natural language similarity
  - Bracketing for ease of processing
  - Push-down automata friendly techniques: limited look-ahead parsing
  - (Advanced techniques: dynamic programming, backtracking, arbitrary-lookahead – not covered)
  - Visual program composition