Domain Specific Language Design (2IMP20)

Loek Cleophas

Ivan Kurtev

Department of Mathematics and Computer Science — Software Engineering and Technology cluster

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Defining DSLs — Grammarware vs. Modelware

Grammarware:

- Language definitions based on *lexical and context-free syntax definitions*

Modelware:

- Language definitions based on *meta-models*

Defining DSLs in the grammar world

Goal: *defining languages* and *manipulation of programs*

Rascal language and environments

- Eclipse based IDE for defining languages, see www.rascal-mpl.org
- An interactive development environment *for defining formal languages and generating tools from them*

Defining DSLs in the grammar world

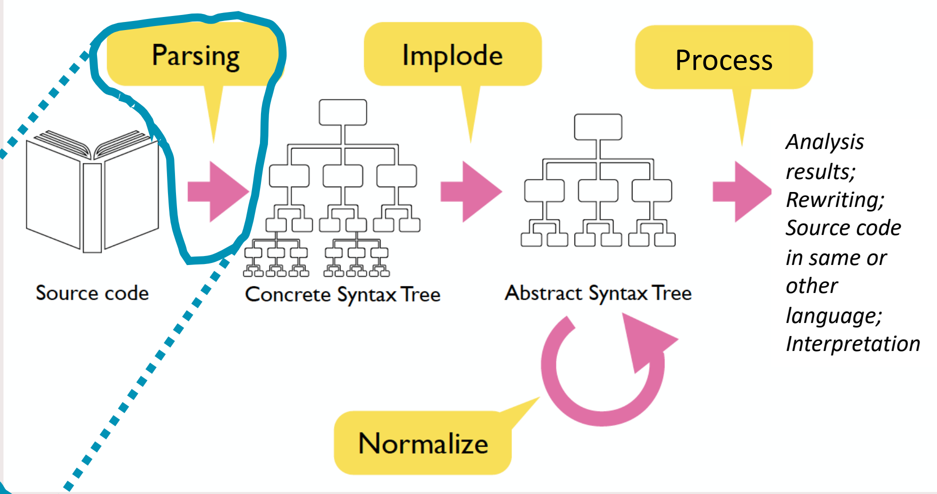
Reading material:

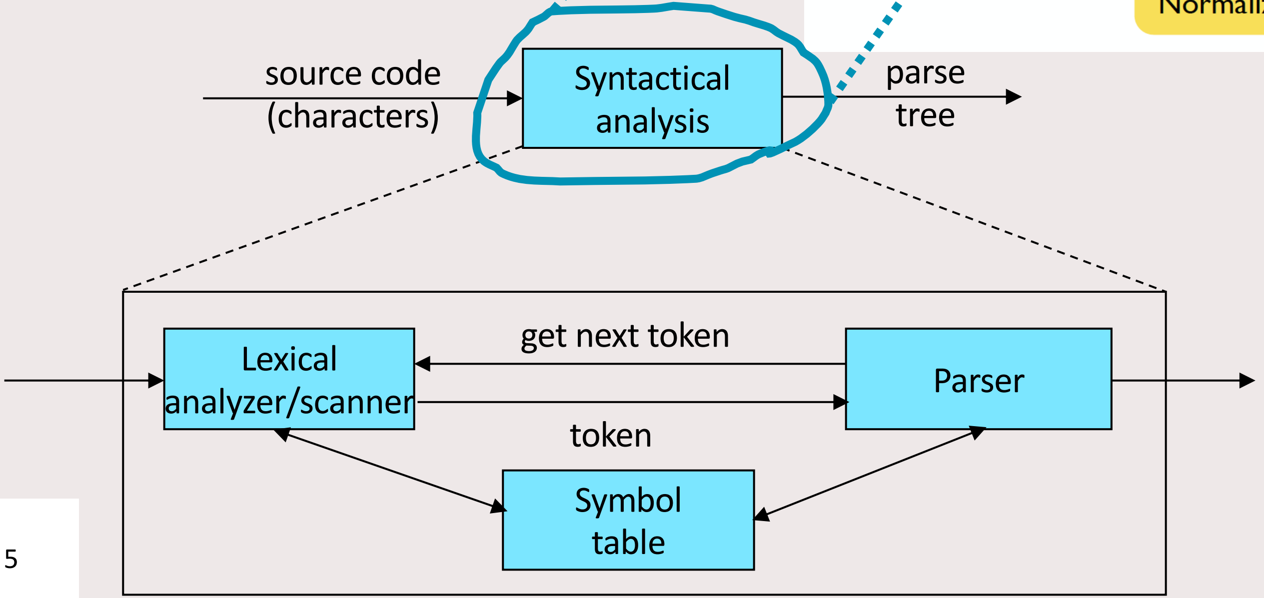
- Chapter 1 of "Introduction to Compiler Design" by Torben Mogensen (see https://link.springer.com/book/10.1007/978-0-85729-829-4, via TU/e library)
- Online material on Rascal: https://www.rascal-mpl.org/docs/Rascal/Declarations/SyntaxDefinition/ (*and see first lecture's slides*)

This + next lecture by Loek:

- Syntax definition and syntactical analysis
  - Lexical + context-free syntax
  - Concrete + abstract syntax

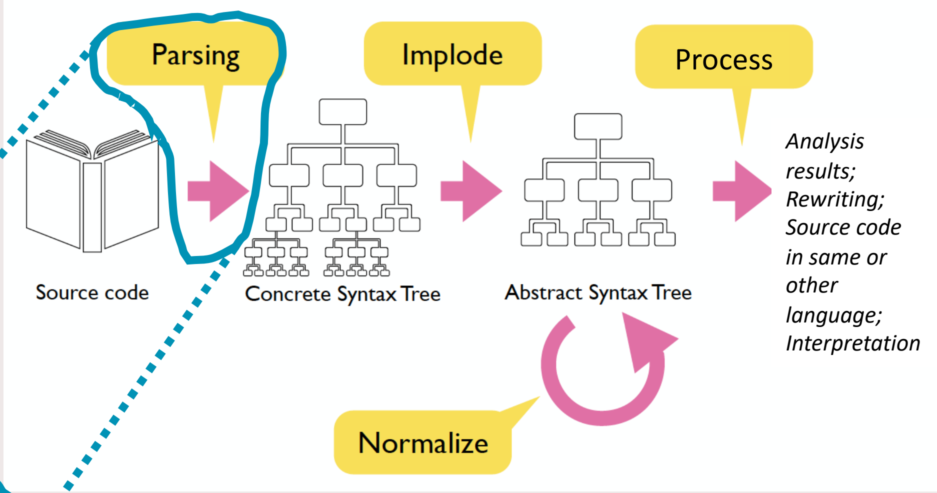
Syntactical analysis

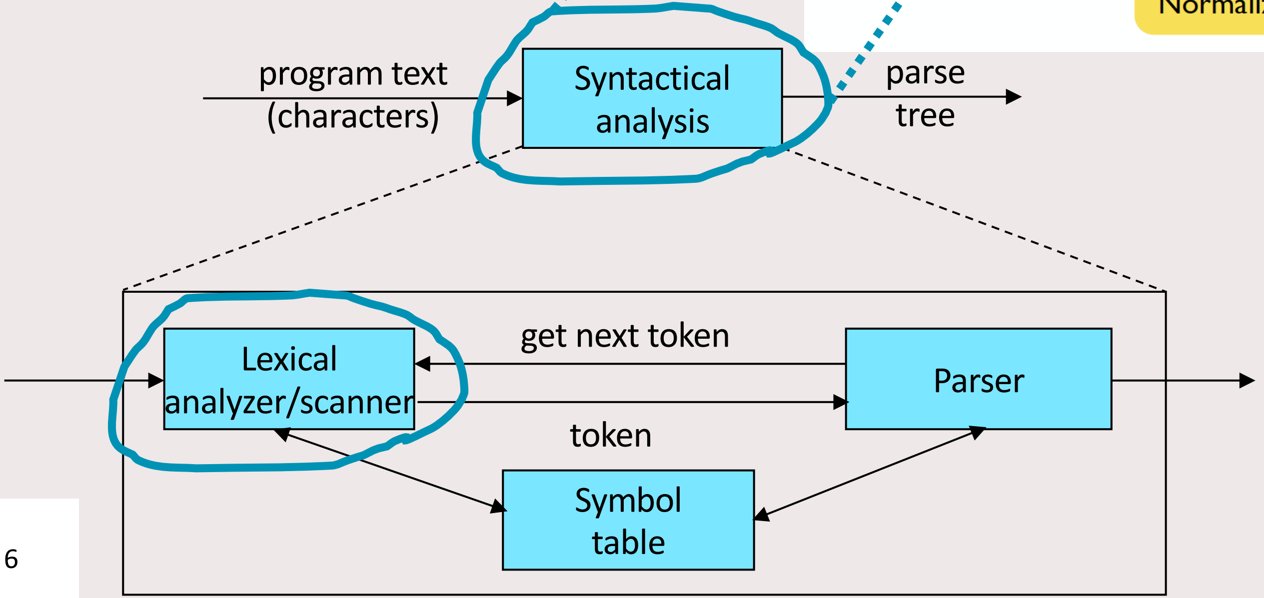




Lexical syntax

- Tasks and organization of *lexical analyzer* a.k.a. *scanner*
- Specification of lexical *tokens* via regular expressions





Lexical syntax

Tasks of the lexical analyzer:

- reading the input and production of tokens
  - each token represents one *logical piece of the source* – keyword, name of a variable, etc. (corresponds to a *syntactic category*)
  - each token may have *optional attributes*
    - including e.g. a *lexeme* (e.g. “myVar”, “+”) ()

- elimination of layout and comments?
- keeping track of position information?

Lexical syntax

What are relevant tokens?

```
for (int k = 0; k < myArray[5]; ++k) {
 cout << k << endl;
}
```

```
for              {
int              }
<<               ;
=                <
(                [
)                ]
++

Identifier
IntegerConstant
```

Lexical syntax

Scanning is not trivial, especially in “old” languages

- FORTRAN: whitespace is irrelevant

```
DO I = 1,25
DOI = 1,25
DOI = 1.25
```

- First and second line represent a loop running from i = 1 to 25
- Third line is an assignment of the value 1.25 to the variable DOI
- Difficult to tell how to partition the input

Lexical syntax

Tokens categorize *lexemes* by what information they represent
(→ notion of *syntactic category*)

Some tokens might be associated with only a single lexeme:

- Tokens for keywords like **if** and **while** probably only match those lexemes exactly

Some tokens might be associated with lots of different lexemes:

- All variables, all possible numbers, all possible strings, etc.

Lexical syntax – How to define?

Regular expressions are a family of descriptions that can be used to capture a class of languages (*regular languages*)

- Provide a compact and human-readable description of the language
- Often used as basis for software systems, e.g. `(f)lex`

Lexical syntax

A regular expression (RE) r over an alphabet Σ corresponds to language $L(r)$

1. ε is a RE and corresponds to $\{\varepsilon\}$
2. $a \in \Sigma$ is a RE and corresponds to $\{a\}$
3. Suppose r and s are REs corresponding to the languages $L(r)$ and $L(s)$
   a. alternative $(r) \mid (s)$ is a RE $\Leftrightarrow L(r) \cup L(s)$
   b. concatenation $(r)\,(s)$ is a RE $\Leftrightarrow L(r)\,L(s)$
   c. Kleene closure $(r)^*$ is a RE $\Leftrightarrow (L(r))^*$
   d. plus $(r)^+$ is a RE $\Leftrightarrow (L(r))^+$
   e. optional $(r)?$ is a RE $\Leftrightarrow L(r) \cup \{\varepsilon\}$
   f. brackets (r) is a RE $\Leftrightarrow L(r)$

Operators are left-associative and priorities are ? > $\{*, {}^+\}$ > concatenation > |

Lexical syntax

Extension of regular expressions:

1. a | b | c | d | … | z $\in \Sigma$ is a RE and corresponds to {a, b, c, d, …, z}, then this is abbreviated as [a-z] (a so-called *character class*); also e.g. [+\–]
2. A *regular definition* over alphabet Σ has the form:

   $\mathtt{id}_1$ ::= $\mathtt{re}_1$
   $\mathtt{id}_2$ ::= $\mathtt{re}_2$
   …
   $\mathtt{id}_n$ ::= $\mathtt{re}_n$

   where id_i are different names and each re_i is a RE over alphabet

   $$\Sigma \cup \{\mathtt{id}_1, \mathtt{id}_2, \ldots, \mathtt{id}_{i-1}\}$$

   Thus, in $\mathtt{re}_i$ only names occur which are already defined

Lexical syntax

Floating point numbers specified in Rascal

```
lexical UnsignedInt = [0] | ([1-9][0-9]*);

lexical SignedInt = [+\-]? UnsignedInt;

lexical UnsignedReal = UnsignedInt [.] [0-9]+ ([eE] SignedInt)?;
lexical UnsignedReal = UnsignedInt [eE] SignedInt;

lexical Number = UnsignedInt | UnsignedReal;
```

```
0    1    14    0.1    3e4    3.014e-7
```

```
00    01    04.1    3e04    3.14e-07
```

Lexical syntax – Challenges in scanning

- How do we determine which lexemes are associated with which token?
- When there are *multiple ways* we could scan the input, how do we know which one to pick?
- ➔ Lexical ambiguities

```
T_For          for
T_Identifier   [A-Za-z\_][A-Za-z0-9\_]*
```

  - How to classify `fort` ?

TU/e

Lexical syntax

Lexical ambiguities: *conflict resolution*

- Assume all tokens are specified as *regular expressions*.
- Algorithm: Left-to-right scan.
- Always *match the longest possible prefix* of the remaining text.

- When two regular expressions apply, *choose the one with higher "priority"*.
  - Simple priority system: pick the rule that was defined first (used by `(f)lex`).

Lexical syntax

Resolution of ambiguities

- Longest match is preferred
- If two alternatives recognize the same sequence of characters, the alternative occurring first in the specification is chosen

```
BEGIN                              [sym := beginsym]
IF                                 [sym := ifsym]
…
letter (letter | digit)*           [sym := idsym]
digit (digit)*                     [sym := intrepsym]
:=                                 [sym := becomessym]
```

Lexical syntax

Rascal offers the class: `lexical` and `keyword`

Repeat zero (*) or one (+) or more times

```
lexical Id = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]) \ Keywords;
keyword Keywords = "if" | "then" | "else" | "fi"
```

A character class: `Id` starts with a lowercase letter

Lexical syntax

Rascal offers the following lexical disambiguation:

- **longest match**
- **prefer keywords**

```
lexical Id = ([a-z0-9] !<< [a-z][a-z0-9]* !>> [a-z0-9]) \ Keywords;
keyword Keywords = "if" | "then" | "else" | "fi"
```

A lexical restriction: is aaa three, two or one identifier? !>> can be used to define longest match. !<< prohibits that an identifier is preceded by characters.

Lexical syntax

- *White space and comments* are serious challenges when defining the lexical syntax
- Rascal *inserts layout* between the elements in the right hand side of a production rule (see next slides)
- In some languages white space has semantics, e.g. Python, Haskell, COBOL, etc.
    - How?
- Comments may be *context-free* instead of regular, for instance, *nested comments*
    - Give a language that supports nested comments

TU/e

Lexical syntax

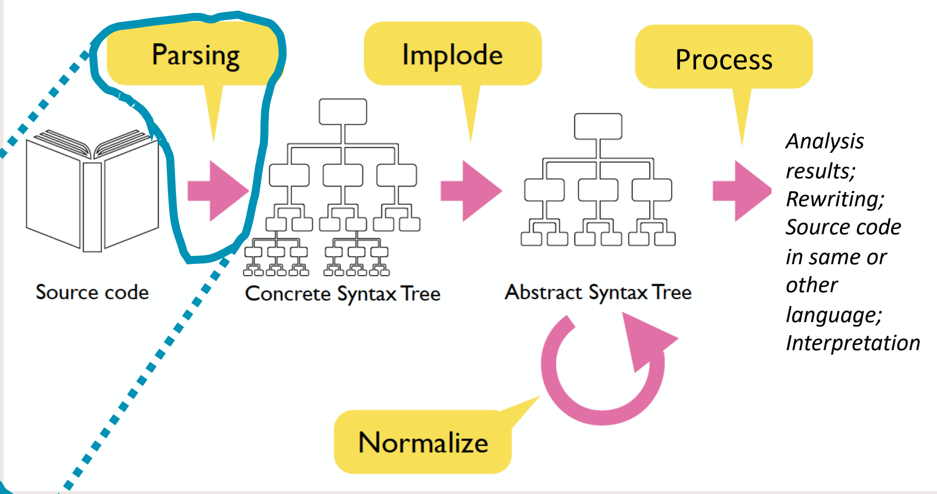
Rascal offers a special lexical class: `layout`

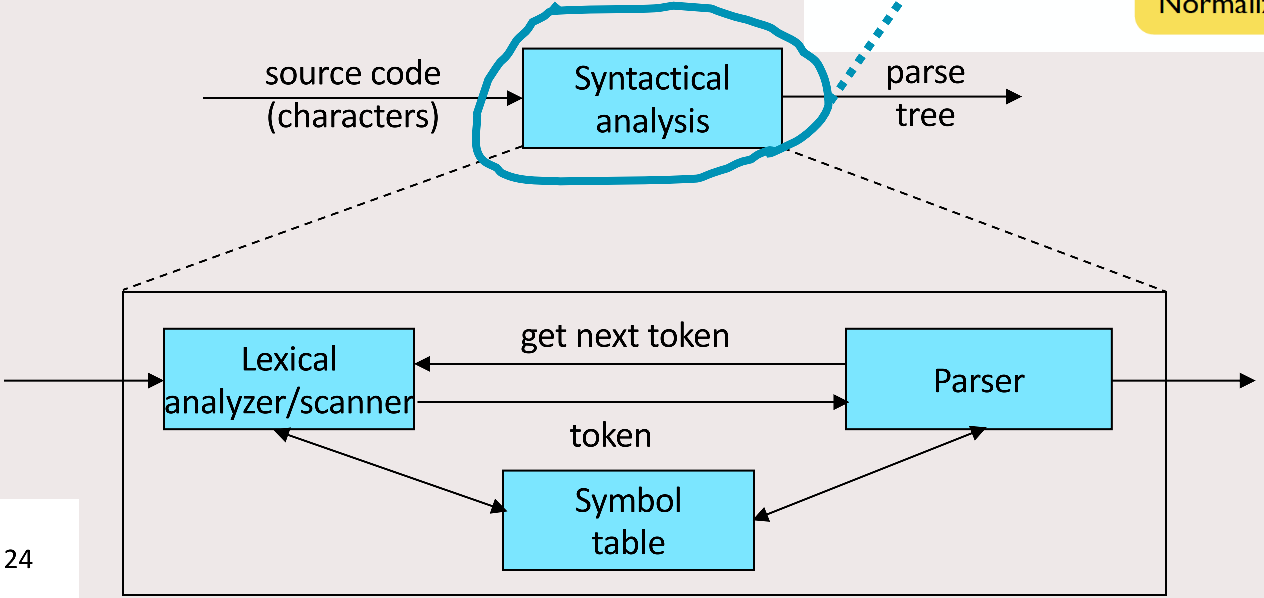
```
layout Layout = WhitespaceAndComment* !>> [\ \t\n\r%];
lexical WhitespaceAndComment =
     [\ \t\n\r]
   | "%" ![%]+ "%"
   | "%%" ![%]* $ ;
```

"$" represents "end of line"

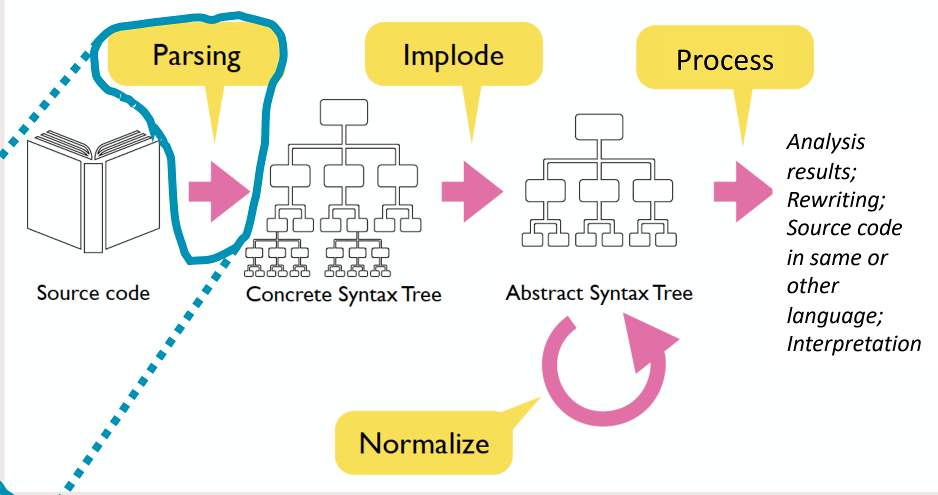
Questions?

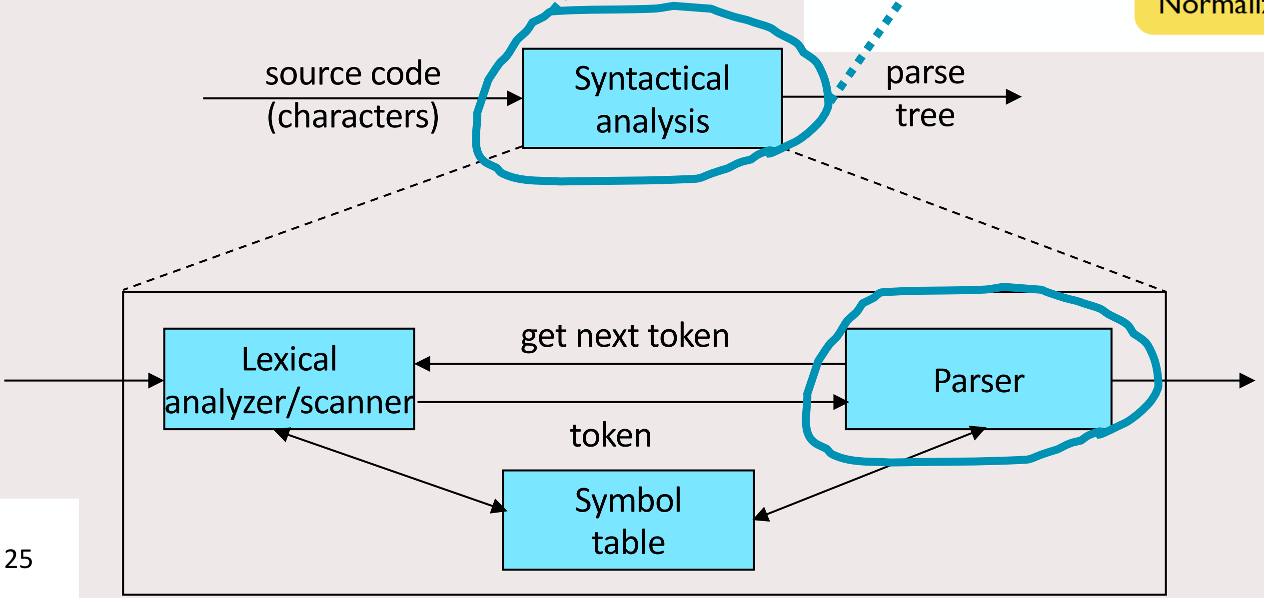
Context-free syntax





Context-free syntax





TU/e

Context-free syntax

Reading material:

- Chapter 2 (“Syntax analysis”) of Introduction to Compiler Design by Torben Mogensen (see https://link.springer.com/book/10.1007/978-0-85729-829-4, via TU/e library)

- **Chapter 7 (“Implementation of Textual Concrete Syntax”) of Software Languages by Ralf Lämmel**

- Online material on Rascal: https://www.rascal-mpl.org/docs/Rascal/Declarations/SyntaxDefinition/ (*and see first lecture’s slides*)

Context-free syntax

What is syntactical analysis?

- After lexical analysis (scanning), we have a *series of tokens*.
- In *syntax analysis (or parsing)*, we want to interpret what those tokens mean.
- Two goals:
  - *Recover the structure described* by that series of tokens.
  - *Report errors* if those tokens do not properly encode a structure.

Context-free syntax

Limits of regular languages

- When scanning, we used regular expressions to define each token
- Unfortunately, regular expressions are (usually) *too weak to define programming languages*.
  - Cannot define a regular expression matching all expressions with *properly balanced parentheses*
  - Cannot define a regular expression matching all functions with *properly nested block structure*
- We need a more powerful formalism

Context-free syntax

A *context-free grammar* is a 4-tuple $G = (N, \Sigma, P, S)$

1. N is a set of *non terminals*

2. Σ is a set of *terminals* (disjoint from N)

3. P is a subset of $N \times (N \cup \Sigma)^*$

   An element $(A, \alpha) \in P$ is called a *production (rule) a.k.a. rule*

   $A ::= \alpha$

4. $S \in N$ is the *start symbol*

The sets N, Σ, P are finite

Context-free syntax

A context-free grammar can be considered as a simple rewrite system with *rewrite steps*: $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A ::= \gamma \in P$ $(\alpha, \beta, \gamma \in (N\cup\Sigma)^*, A \in N)$

NB: this covers case S ⇒ …

Example $N = \{\mathtt{E}\}$, $\Sigma = \{\mathtt{+}, \mathtt{*}, \mathtt{(}, \mathtt{)}, \mathtt{-}, \mathtt{a}\}$, $S = \mathtt{E}$,

P = {
```
E ::= E + E
E ::= E * E
E ::= ( E )
E ::= - E
E ::= a
```
}

Derivation `E ⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(a+E) ⇒ -(a+a)`

Context-free syntax

The *language* $L(G)$ generated by the context-free grammar
$G = (N, \Sigma, P, S)$ is: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^+ w\}$

A *sentence* $w \in L(G)$ contains *only terminals*

A *sentential form* α is a string of terminals and non-terminals
which can be derived from S:

$$S \Rightarrow^* \alpha \text{ with } \alpha \in (N \cup \Sigma)^*$$

→ A *sentence* in $L(G)$ is a sentential form in which *no* non-terminals occur

Context-free syntax

Derivations

- A sequence of steps is called a *derivation*.
- A string $\alpha A \omega$ *yields* string $\alpha\gamma\omega$ iff $A \rightarrow \gamma$ (a.k.a. $A ::= \gamma$) is a production.
  - If α yields β, we write $\alpha \Rightarrow \beta$.
- We say that α *derives* β iff there is a sequence of strings where
  $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \beta$
  - If α *derives* β, we write $\alpha \Rightarrow^* \beta$.

Context-free syntax

Left/right derivations

- There are choices to be made for each derivation step:
    - *which non-terminal* must be replaced?
    - *which alternative of the selected non-terminal* (i.e. *which rule*) must be applied?

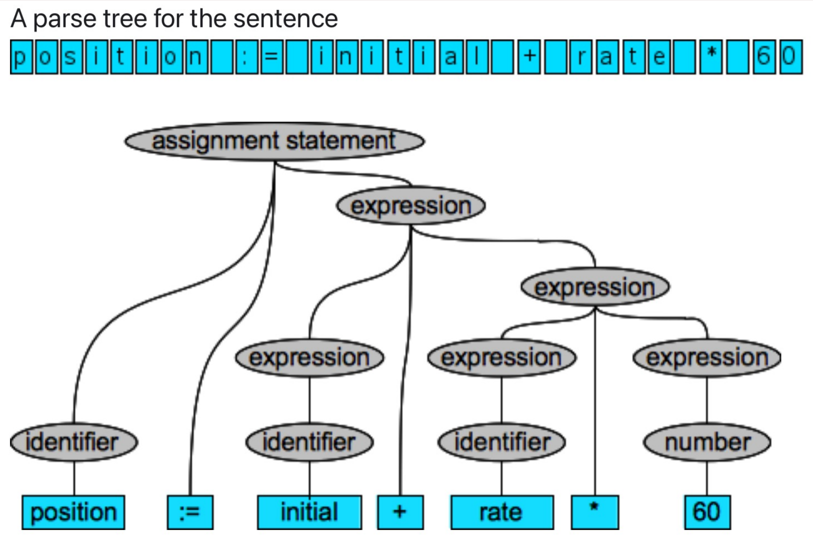
- *Always* selecting the leftmost non-terminal in the sentential form gives a *leftmost derivation*: $\Rightarrow_{lm}$
    - There exists also a *rightmost* derivation: $\Rightarrow_{rm}$

- Consider the context-free grammar for expressions:
    - Leftmost derivation for `-(a+a)` : $\texttt{E} \Rightarrow \texttt{-E} \Rightarrow \texttt{-(E)} \Rightarrow \texttt{-(E+E)} \Rightarrow \texttt{-(a+E)} \Rightarrow \texttt{-(a+a)}$
    - Rightmost derivation for `-(a+a)` : $\texttt{E} \Rightarrow \texttt{-E} \Rightarrow \texttt{-(E)} \Rightarrow \texttt{-(E+E)} \Rightarrow \texttt{-(E+a)} \Rightarrow \texttt{-(a+a)}$

Context-free syntax

Parse trees

- A parse tree is a *tree encoding the steps in a derivation*.
  - *Internal nodes* represent *nonterminal symbols* used in the production.
  - *In order walk of the leaves* contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied



(Source: https://www.rascal-mpl.org/docs/Rascalopedia/ParseTree/)

TU/e

Context-free syntax

A parse tree for a context-free grammar $G = (N, \Sigma, P, S)$ is a tree:

1. The *root* is labeled with S (the start non-terminal)
2. Each *leaf* is labeled with a terminal ($\in \Sigma$) or ε
3. All *other nodes* are labeled with a non-terminal

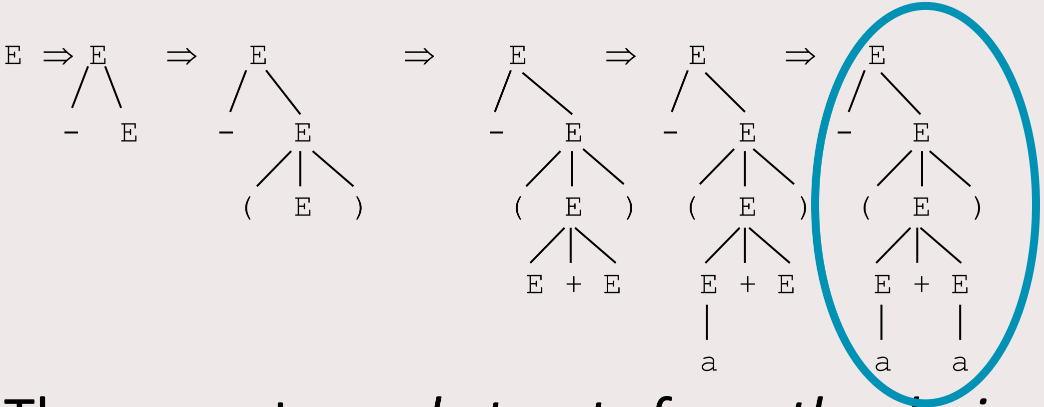
If A is the label of a node and $X_1,\ldots,X_n$ are the labels of the children (from left to right) then $A ::= X_1,\ldots,X_n$ **must** be a production rule in G (where each X_i is either a terminal or a non-terminal)

Special case is $A ::= \varepsilon$ with label A which has exactly one child with label ε, also called a *nullable* non-terminal

Context-free syntax

Example:

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(a+E) \Rightarrow -(a+a)$



The parse tree *abstracts from the derivation order*

Context-free syntax

Acceptor and parser

For each grammar G there exists a decision procedure (acceptor) AG for $L(G)$:

$$AG: STRING \rightarrow \{true, false\}$$

such that

$$AG(w) = true \Leftrightarrow w \in L(G)$$

A *parser* is an acceptor which constructs a parse tree as well.

- A *top-down* parser constructs the tree starting from the root
- A *bottom-up* parser constructs the tree starting from the leaves

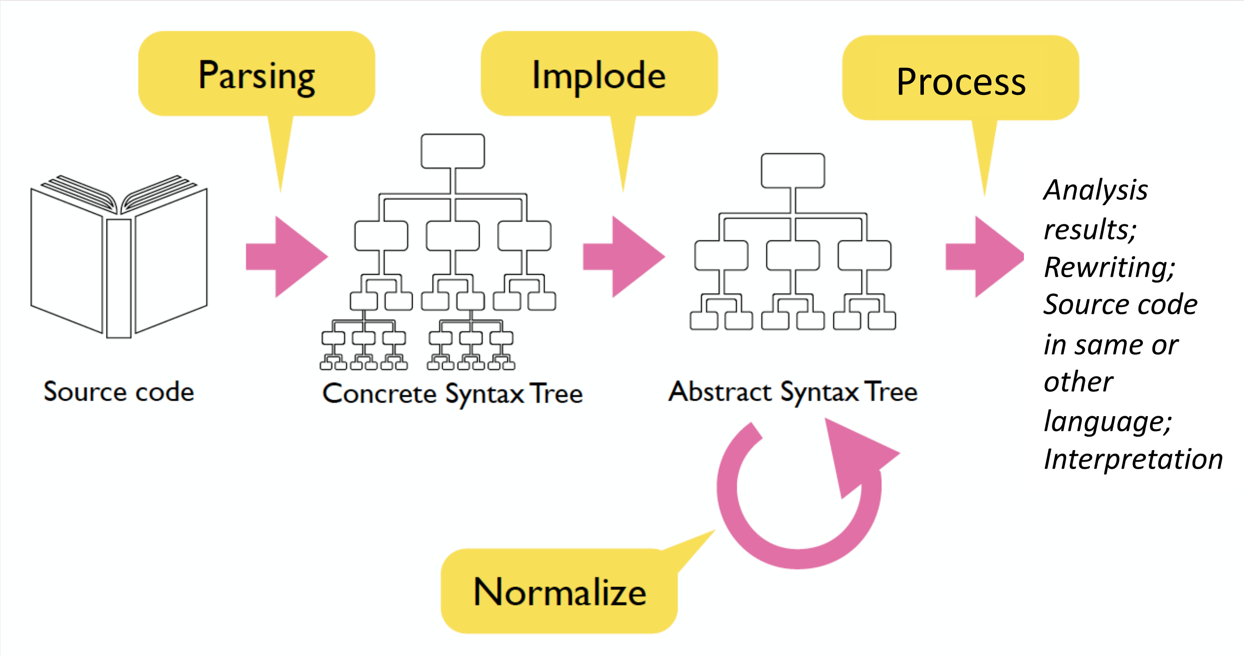
Context-free syntax

Goals of parsing

- *Recover the structure* described by a series of tokens.

- If language is described as a CFG, goal is to *recover a parse tree for the input string*.
- If the input string is syntactically incorrect: *generate an error message*

Why a parse tree?



Context-free syntax

During parsing the following problems may occur:

- The grammar is ambiguous
- The grammar is left recursive
- The grammar contains cycles
- Grammars are in principle not modular

Context-free syntax

Syntax definitions in Rascal

- Rascal offers a modular way of defining the syntax of a language

```
module demo::lang::Pico::Syntax

import Prelude

…
```

Context-free syntax

name of the abstract syntax tree node

Syntax definitions in Rascal

- The start non-terminal is explicitly defined as follows

```
start syntax Program =
   program: "begin" Declarations decls {Statement ";"}* body "end";

…
```

name of the attributes of the AST node

Context-free syntax

Syntax definitions in Rascal

- Normal production rules:

```
…
syntax Declarations =
   "declare" {Declaration ","}* decls ";" ;

syntax Declaration = decl: Id id ":" Type tp ;

syntax Type =
    natural: "natural"
  | string: "string" ;

…
```

Context-free syntax

Syntax definitions in Rascal

- Normal production rules:

```
…
syntax Statement =
  asgnStat: Id var ":=" Expression val
  | ifElseStat: "if" Expression cond "then" {Statement ";"}* thenPart
                   "else" {Statement ";"}* elsePart "fi" ;
  | whileStat: "while" Expression cond "do" {Statement ";"}* body "od" ;

…
```

Context-free syntax

Syntax definitions in Rascal

- Normal production rules:

disambiguation via priorities of production rules

```
…
syntax Expression =
  id: Id name
  | strCon: String string
  | natCon: Natural natural
  | bracket: "(" Expression expr ")" ;
  > left conc: Expression lhs "||" Expression rhs
  > left ( add: Expression lhs "+" Expression rhs
         | sub: Expression lhs "-" Expression rhs
         ) ;
…
```

disambiguation via left associativity of binary operators

Questions?

Context-free syntax

During parsing the following problems may occur:

- The grammar is ambiguous
- The grammar is left recursive
- The grammar contains cycles
- Grammars are in principle not modular

When is a grammar ambiguous?

A grammar G is *ambiguous* if one word $w \in L(G)$ has at least two parse trees

- Expression grammar without associativities and priorities
- Dangling else problem

Context-free syntax

Ambiguity

- A CFG is said to be ambiguous if there is at least one string with two or more parse trees.
- Note that *ambiguity is a property of grammars*, not languages.
- There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.
    - Some languages are inherently ambiguous, meaning that no unambiguous grammar exists for them.
- There is no algorithm for detecting whether an arbitrary grammar is ambiguous.

Context-free syntax

Is ambiguity a problem?

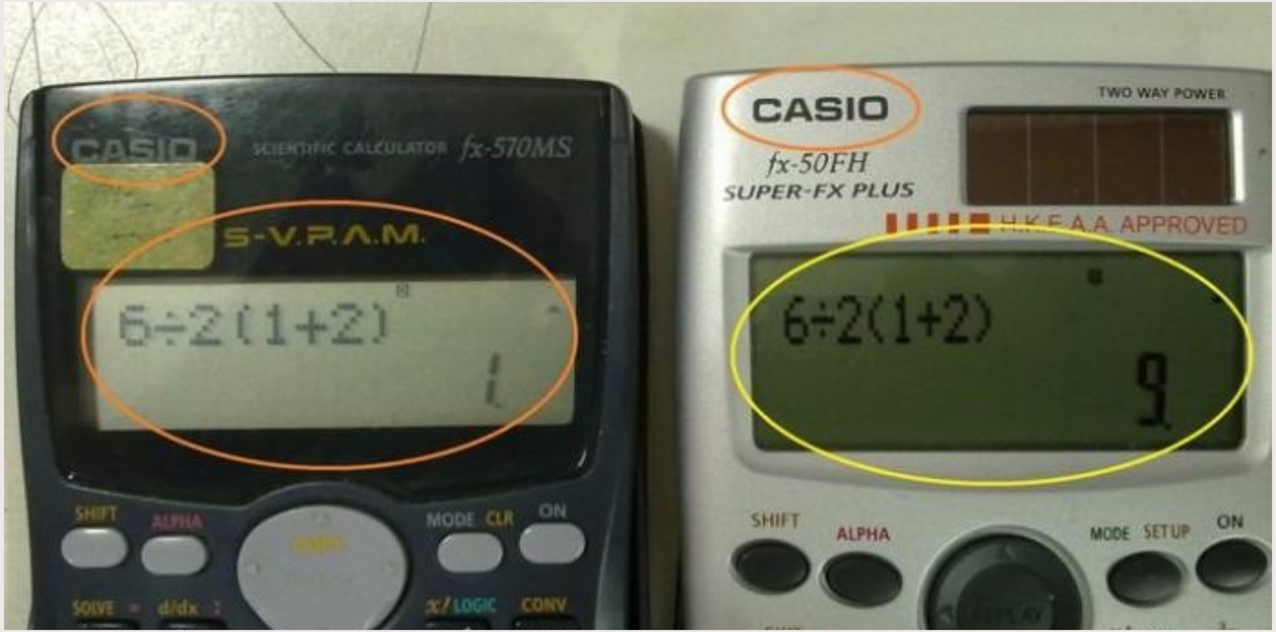
- depends on semantics

```
E ::= int | E + E

  1 + 2 + 3
```

vs

```
E ::= int | E + E | E - E

  5 - 3 + 7
```

Context-free syntax



Context-free syntax

Resolving ambiguities

- If a grammar can be made unambiguous at all, it is usually made unambiguous through *layering*.
    - Have exactly *one way to build each piece* of the string.
    - Have exactly *one way of combining those pieces* back together.

Context-free syntax

Example of balanced parentheses

- Consider the language of all strings of balanced parentheses:
  - ε
  - ( )
  - (( ) ( ))
  - ((( ))) (( )) ( )
- A possible grammar for balanced parentheses is:

```
S ::= ε | S S | ( S )
```

- Why is this grammar ambiguous?
- How to resolve this ambiguity?

Context-free syntax

Restructuring the grammar of parentheses

- A string of balanced parentheses is a sequence of strings that are themselves balanced parentheses
- To avoid ambiguity, we can build the string in two steps:
  - Decide how many different substrings we will glue together.
  - Build each substring independently.

Context-free syntax

Building a new grammar for parentheses

- Spread a string of parentheses across the string. There is exactly one way to do this for any number of parentheses.
- Expand out each substring by adding in parentheses and repeating.

Another grammar for balanced parentheses is:

```
S ::= P S | ε
P ::= ( S )
```

Context-free syntax

Why explicit disambiguation in Rascal?

- Fully declarative definition of syntax
- Implicit disambiguation can give unexpected results
- More programming languages can be parsed:
  - legacy languages
- Modularity
- Fewer non-terminals/sorts
- Separation of concerns: form of rules is independent of disambiguation

Questions?

Context-free syntax

During parsing the following problems may occur:

- The grammar is ambiguous
- The grammar is left recursive
- The grammar contains cycles
- Grammars are in principle not modular

Context-free syntax

Left recursion removal:

- A grammar is *immediate left recursive* if the grammar contains a rule of the form $A ::= A\alpha$

- A grammar is *left recursive* if there exists a non-terminal *A* and a string $\alpha \in (N \cup \Sigma)^*$ such that $A \Rightarrow^* A\alpha$
- This means that after one or more steps in a derivation an occurrence of *A* reduces again to an occurrence of *A* without recognizing any terminal in the input sentence.

Context-free syntax

Examples of *indirect* left recursion

$A ::= B\ \alpha$
$B ::= A\ \beta$

or worse

$A ::= B\ \alpha$
$B ::= D\ A\ \beta$
$D ::= \varepsilon \mid \gamma\ G$

It is easy to remove left recursion from a context-free grammar

TU/e

Questions?