Regular languages

Definition

A regular language is a language definable with a regular expression.

Remark: regular languages can be defined in other equivalent ways:

- with *right* or *left regular* (also called *linear*) grammars;
- with non-deterministic or deterministic finite automata (NFA or DFA).

Limitations

Regular languages are too simple.

Examples of regular languages:

- the language of identifiers
- the language of numbers (integer radix 2, 8, 10, 64, floating-point)
- the language of string constants

Remark: regular expressions are useful to define the syntax of the lexemes of a programming language, but cannot define the syntax of the full language.

Regular languages

Examples of non-regular languages

- The language of expressions with numbers, binary addition and multiplication and parentheses cannot be defined by a regular expression.

  `{"0","12","2+5","(2+5)*3",...}`

  Remark: the actual problem are the parentheses; if one removes them, then the language becomes regular!
- A very simple example of non-regular language:

  $\{\mathtt{a}^n\mathtt{b}^n \mid n \text{ natural number}\} = \{\texttt{""},\texttt{"ab"},\texttt{"aabb"},\texttt{"aaabbb"},\ldots\}$.

Syntax analysis of a program

Syntax analysis defined on top of lexical analysis

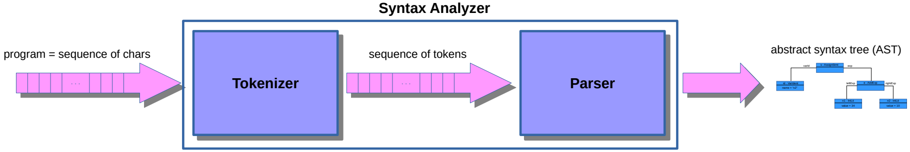
Syntax analysis is the problem of

- recognizing valid sequences of tokens of a program by following the syntactic rules of the language
- building, in case of success, an Abstract Syntax Tree (AST)
  - AST = an abstract representation of the syntax of the recognized program
  - an AST makes explicit the structure of the syntax: statements/expressions are built on simpler sub-statements/sub-expressions
  - AST = input to the other steps of a programming language implementation:
    - typechecking specified by the static semantics
    - interpretation/compilation specified by the dynamic semantics

Parser

A program which performs syntax analysis

Syntax analysis of a program



Parser for a programming language

- input: sequence of tokens of a program, recognized by a tokenizer
- it checks that the sequence of tokens verifies the syntax rules
- the syntax rules are formally defined by a grammar
- output:
- an Abstract Syntax Tree (AST)
- it can be hand-written or automatically generated by an application (ANTLR, Bison, ...)

A parser at work

Example 1 with C/Java/C++/C# syntax

Input string: `"x2 042=;"`

Analyzed tokens:

- `IDENTIFIER` with syntactic data: the name `"x2"`
- `INT_NUMBER` with semantic data: the value thirty-four
- `ASSIGN_OP` with no further data
- `STATEMENT_TERMINATOR` with no further data

Result of the parser

failure, the sequence is not recognized and error messages are reported

A parser at work

Example 2 with C/Java/C++/C# syntax

Input string: `"x2=042+012;"`

Analyzed tokens:

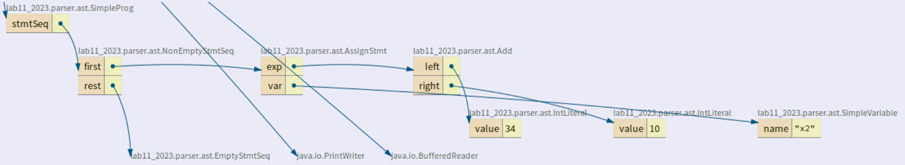
- `IDENTIFIER` with syntactic data: the name `"x2"`
- `ASSIGN_OP` with no further data
- `INT_NUMBER` with semantic data: the value thirty-four
- `ADD_OP` with no further data
- `INT_NUMBER` with semantic data: the value ten
- `STATEMENT_TERMINATOR` with no further data

Result of the parser

success, the sequence is recognized and an AST is generated (see next slide)

A parser at work

Result of the parser: an AST



Context free (CF) grammars

CF grammars for programming languages

- standard formalism to define the syntax of programming languages
- more expressive than regular expressions
  - basic operators: concatenation and union (as regular expressions)
  - more powerful feature:
    Kleene star (=iteration) replaced by recursion (=induction)

A BNF grammar (Backus-Naur Form or Backus Normal Form)

A CF grammar in BNF syntax to define a simple language of expressions

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Remark

- here `Num` is defined in the grammar only for educational aim
- in practice, `Num` is defined separately by a regular expression

Context free (CF) grammars

Revisited example used in practice

```
Exp ::= NUM | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
```

`NUM` is defined, for instance, by the regular expression `0|1`

We follow the following notation

- in `Exp` only the first letter is capitalized:
  it means that `Exp` is defined in the grammar
- in `NUM` all letters are capitalized:
  it means that
  - `NUM` corresponds to a set of lexems (= a token type)
  - `NUM` is defined separately by a regular expression

Terminology of CF grammars

Grammar

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Terminology: grammar $G = (T, N, P)$

- $\{\texttt{'+'}, \texttt{'*'}, \texttt{'('}, \texttt{')'}, \texttt{'0'}, \texttt{'1'}\}$ is the set T of terminal symbols
- $\{\texttt{Exp}, \texttt{Num}\}$ is the set N of non-terminal symbols
- $\{(\texttt{Exp},\texttt{Num}), (\texttt{Exp},\texttt{Exp '+' Exp}), (\texttt{Exp},\texttt{Exp '*' Exp}), (\texttt{Exp},\texttt{'(' Exp ')'}), (\texttt{Num},\texttt{'0'}), (\texttt{Num},\texttt{'1'})\}$ is the set P of productions

Remarks

- each non terminal corresponds to a language; languages are defined as unions of concatenations
- terminal symbols are lexemes of the languages defined by the grammar
- productions have shape (B, α) where $B \in N$ and $\alpha \in (T \cup N)^*$

Grammars as inductive definitions of languages

Grammar

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Inductive definition of languages

$$Exp = Num \cup (Exp \cdot \{\texttt{"+"}\} \cdot Exp) \cup (Exp \cdot \{\texttt{"*"}\} \cdot Exp) \cup (\{\texttt{"("}\} \cdot Exp \cdot \{\texttt{")"}\})$$
$$Num = \{\texttt{"0"}\} \cup \{\texttt{"1"}\}$$

Remarks

- *Num* is the base case for *Exp*: a number is an expression
- *Exp* is defined on top of *Num*, *Num* is defined only by base cases

Grammars as inductive definitions of languages

Another grammar

```
Exp ::= Term | Exp '+' Term | Exp '*' Term
Term ::=  '(' Exp ')' | Num
Num ::= '0' | '1'
```

Remarks

The definitions of *Exp* and *Term* are mutually recursive

Derivations

Grammar

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Languages generated by a grammar

- A grammar generates a language for each non-terminal symbol
- The grammar above generates the two languages L_{Exp} and L_{Num}
- The language for `Num` is pretty simple: $L_{Num} = \{\texttt{"0"},\texttt{"1"}\}$

Questions

- How is L_{Exp} defined?
- How can we show that `"0*1"` $\in L_{Exp}$ and `"1+*("` $\notin L_{Exp}$

Answer: derivations in one or more steps are used

One-step derivation

Grammar

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Example of one-step derivations

$\underline{Exp} \rightarrow Exp * Exp$	production (Exp, $Exp*Exp$) is used
$\underline{Exp} * Exp \rightarrow Num * Exp$	production (Exp, Num) is used
$Num * \underline{Exp} \rightarrow Num * Num$	production (Exp, Num) is used
$\underline{Num} * Num \rightarrow 0 * Num$	production (Num, 0) is used
$0 * \underline{Num} \rightarrow 0 * 1$	production (Num, 1) is used

Remarks

- no other derivation steps from 0 * 1 (no production can be used)
- 0 * 1 belongs to L_{Exp}

Definition of derivation

One-step derivation $\rightarrow$

One-step derivation for a grammar $G = (T, N, P)$

- it has shape $\alpha_1 B \alpha_2 \rightarrow \alpha_1 \gamma \alpha_2$
- $\alpha_1, \alpha_2 \in (T \cup N)^*$
- $(B, \gamma) \in P$ that is, (B, γ) is a production of G

Multi-step derivation $\rightarrow^+$

Transitive closure of $\rightarrow$:

- base case: if $\gamma_1 \rightarrow \gamma_2$, then $\gamma_1 \rightarrow^+ \gamma_2$
- inductive case: if $\gamma_1 \rightarrow \gamma_2$ and $\gamma_2 \rightarrow^+ \gamma_3$, then $\gamma_1 \rightarrow^+ \gamma_3$

Definition of language

Definition of language generated by a grammar

Language L_B generated from $G = (T, N, P)$ for non-terminal $B \in N$

- all strings of terminals that can be derived in one or more steps from B
- formally: $L_B = \{u \in T^* \mid B \rightarrow^+ u\}$