CSE216
Programming Abstractions

State University of New York, Korea

Agenda

- Regular expression
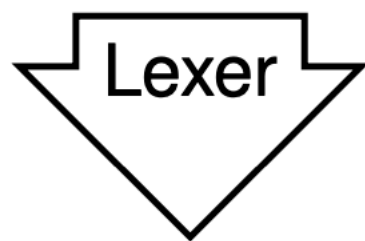
- context free grammar

Regular Expressions

What is a program?

- Let us consider an expression x + y*10

- Think of this expression as a program in a programming language

- This is actually a program written in a programming language used by a calculator

- Today we will analyze the syntax of a general program — Syntax analysis

- Syntax analysis can take a whole semester to learn; we will touch only the surface
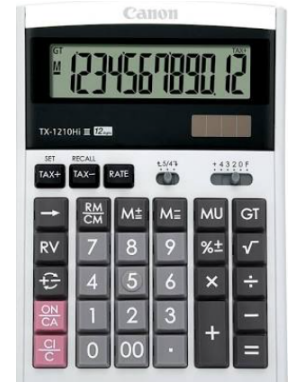
How to specify program syntax?
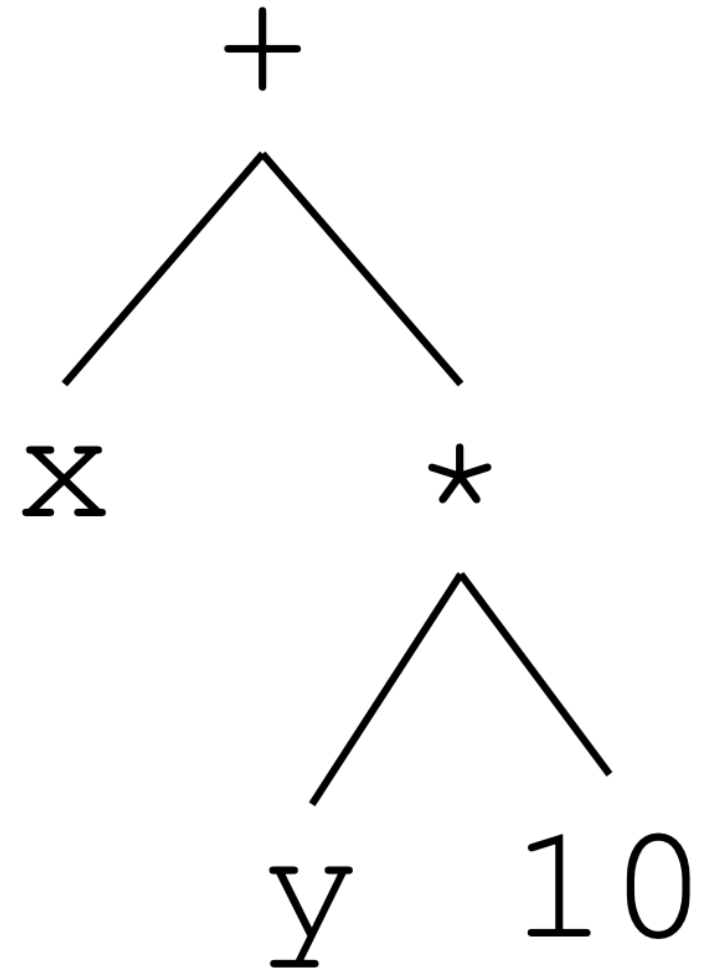
"x +y*10"

Lexer

Regular expression Specification

| x | + | y | * | 10 |

Parser

Context-free grammar specification

```
      +
     / \
    x   *
       / \
      y   10
```

Regular expression specification looks like this in Ocaml

```
rule Token = parse
  | [' ' '\t' '\n' '\r']    { Token lexbuf }
  | ['0'-'9']+              { CSTINT (...) }
  | ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']*
                            { keyword (...) }
  | '+'                     { PLUS  }
  | '-'                     { MINUS }
  | '*'                     { TIMES }
  | '('                     { LPAR  }
  | ')'                     { RPAR  }
  | eof                     { EOF   }
  | _                       { lexerError lexbuf "Bad char" }
```

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"a"\}$
ε	Empty string	$\{""\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"a"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"a"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"a"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \dots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \mid r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
$\texttt{a}$	Character $\texttt{a}$	$\{"\texttt{a}"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \vert r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

Examples

$\quad \texttt{ab}^*$ represents $\{"\texttt{a}","\texttt{ab}","\texttt{abb}",\ldots\}$

$\quad (\texttt{ab})^*$ represents $\{"","\texttt{ab}","\texttt{abab}",\ldots\}$

$\quad (\texttt{a}\vert\texttt{b})^*$ represents $\{"","\texttt{a}","\texttt{b}","\texttt{aa}","\texttt{ab}","\texttt{ba}",\ldots\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"a"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \mid r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

Examples

ab^* represents $\{"a", "ab", "abb", \ldots\}$

$(ab)^*$ represents $\{"", "ab", "abab", \ldots\}$

$(a\mid b)^*$ represents $\{"", "a", "b", "aa", "ab", "ba", \ldots\}$

Exercise

What does $(a\mid b)c^*$ represent?

Regular expression abbreviations

Abbrev.	Meaning	Expansion
`[aeiuo]`	Set	`a｜e｜i｜o｜u`
`[0-9]`	Range	`0｜1｜...｜8｜9`
`[0-9a-z]`	Ranges	`0｜1｜...｜8｜9｜a｜b｜...｜y｜z`
$r?$	Zero or one r	$r\|\varepsilon$
r^+	One or more r	$r\,r^*$

Exercises:
Regular Expression

Exercise 1

Write regular expressions for:

- Non-negative integer constants

Demo: https://regex101.com/

Exercise 2

Write regular expressions for:

- Integer constants

Exercise 3

Write regular expressions for:

- Floating-point constants:
  - 3.14
  - 3E8
  - +6.02E23
  - **3E+08**
  - **4.6E-09**

Exercise 4

Write regular expressions for:

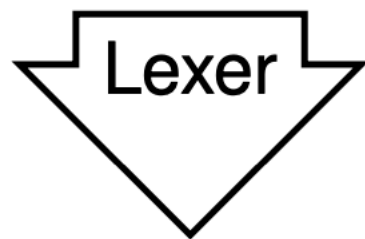- Java variable names:
    - xy
    - x12
    - _x

Exercise 5

Give a RE for: $L = \{0^i 1^j \mid i \text{ is even and } j \text{ is odd }\}$

0^i above refers to repeating "0" i times. E.g. 0^4 means "0000"

Context-free grammar

Parsing

The need for a grammar

"Afternoon good, I'd room a like."

MR. TOPSY-TURVY
By Roger Hargreaves
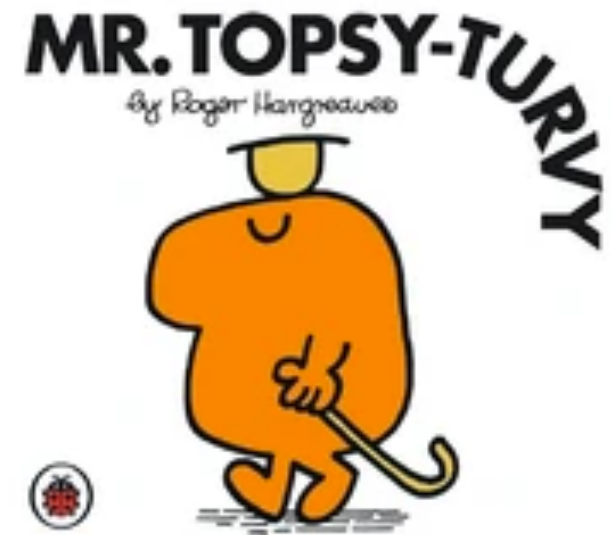
Mr. Men and Little Miss Series

Context-Free Grammar

- A notation for describing languages.

- More powerful than regular expressions.

- It still cannot define all possible languages.

- Useful for recursive structures, e.g., most today's programming languages.

Basic idea

- A language can be decomposed to smaller parts

- Each part can be defined recursively

- Use **production rules** to generate the language

Example 1

```
Program ::= Stmt Program
         | ""
Stmt     ::= Var = AExpr
         | if ( BExpr ) Stmt else Stmt
         | while ( BExpr ) Stmt
AExpr    ::= AExpr + AExpr
         | AExpr - AExpr
         | AExpr * AExpr
         | AExpr / AExpr
         | Var
         | Number
BExpr    ::= AExpr == AExpr
         | AExpr < AExpr
         | not (BExpr)
         | BExpr and BExpr
         ...
Var      ::= x | y | z ...
Number ::= 0 | 1 | 2 ...
```

Demo: Parse tree for x = 2

Example 2

- Production rule:

  - $S \rightarrow 01$

  - $S \rightarrow 0S1$

- Basis: 01 is in the language.

- Induction: If w is in the language, then so is 0w1.

- The generated language is $\{0^n 1^n, n >= 1\}$

Overview

- Use **terminal** symbols a, b, c, d… for the alphabet of a language.

- Use **nonterminal** symbols A, B, C, D, recursively

- **Starting symbol** is a special nonterminal

- Use **production rules** to generate the language

Production

- A production has the form

  variable → string of variables and terminals

- Convention
  - A, B, C,... are variables.
  - a, b, c,... are terminals.

Put Everything Together

- Here is a formal CFG for $\{0^n1^n \mid n \geq 1\}$.
- Terminals $= \{0,1\}$.
- Variables $= \{S\}$.
- Start symbol $= S$.
- Productions $=$
  $S \rightarrow 01$
  $S \rightarrow 0S1$

Notation

- Symbol ::= is often used for →.
- Symbol | is used for or.
  - A shorthand for a list of productions with the same left side.

  Example: $S → 0S1 \mid 01$ is a shorthand for $S → 0S1$ and $S → 01$.

Exercise 1: Construct a parse tree

while (x<10) x = x + 1

```
Program ::= Stmt Program
        | ""
Stmt    ::= Var = AExpr
        | if ( BExpr ) Stmt else Stmt
        | while ( BExpr ) Stmt
AExpr   ::= AExpr + AExpr
        | AExpr − AExpr
        | AExpr * AExpr
        | AExpr / AExpr
        | Var
        | Number
BExpr   ::= AExpr == AExpr
        | AExpr < AExpr
        | not (BExpr)
        | BExpr and BExpr
        ...
Var     ::= x | y | z ...
Number ::= 0 | 1 | 2 ...
```

Exercise 2: Construct a parse tree

```
AExpr    ::= AExpr + AExpr
         | AExpr - AExpr
         | AExpr * AExpr
         | AExpr / AExpr
         | Var
         | Number
BExpr    ::= AExpr == AExpr
         | AExpr < AExpr
         | not (BExpr)
         | BExpr and BExpr
         ...
Var      ::= x | y | z ...
Number ::= 0 | 1 | 2 ...
```

1 + 0 * 2

Exercise 3: Construct a parse tree

```
expr    ::= term
          | expr + term
          | expr – term

term    ::= factor
          | term * factor
          | term / factor

factor ::= NUMBER
          | NAME
          | ( expr )
```

1 + 0 * 2

Ambiguous grammar

- An ambiguous grammar is a formal grammar that can produce **multiple parse trees** or interpretations for the same input sentence or sequence of symbols.

- This can be problematic in various contexts because it can **make it difficult to determine the correct meaning** or parse tree of a sentence or sequence of symbols.

- To avoid ambiguity, it is often necessary to **use unambiguous grammars** or to add rules or constraints to the ambiguous grammar to disambiguate the interpretations.

Summary

- Context free grammar concepts

- Parse tree

- Ambiguous grammar

- Grammar -> Language

- Language -> Grammar