

IN2009  
Language Processors

Week 2  
Language Processing & Lexical Analysis

Igor Siveroni

## Session Plan

### Session 2

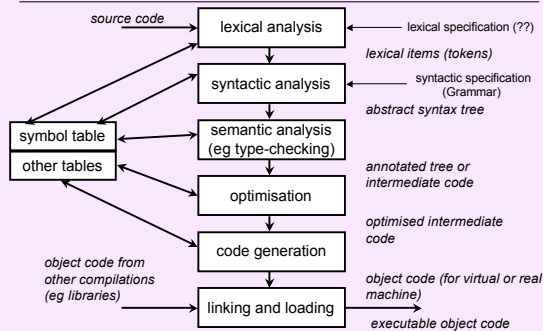
- Language processing
- Lexical analysis
- Syntax analysis
- Lexical Tokens
- Regular expressions
- Finite Automata
- Implementation
- Tools

9 February, 2009

IN2009 Language Processors - Session 2

2

## Language processing



9 February, 2009

IN2009 Language Processors - Session 2

3

## Lexical analysis

### Straightline example program:

```

a := 5+3;
b := (print(a, a-1), 10*a);
print(b)

```

Lexical analysis converts text stream into a token stream, where tokens are the most basic symbols (words and punctuation):

```

a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; ...

```

Each box is a lexical item or token. A possible representation:

```

ID(a) ASSIGN NUM(5) PLUS NUM(3) SEMI ID(b) ASSIGN LEFTPAREN
KEYPRINT LEFTPAREN ID(a) COMMA ID(a) MINUS NUM(1)
RIGHTPAREN COMMA NUM(10) TIMES ID(a) RIGHTPAREN SEMI ...

```

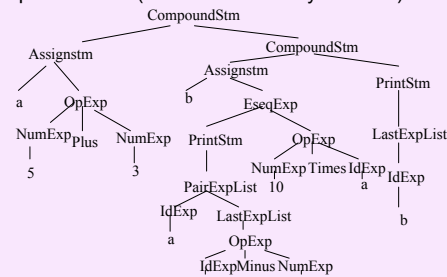
9 February, 2009

IN2009 Language Processors - Session 2

4

## Syntax analysis

Converts a token stream into a useful abstract representation (here an abstract syntax tree):



## Lexical tokens

Type	Examples
ID	foo n14 last
NUM	73 0 00 515 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
LPAREN	(
ASSIGN	:=

- Some tokens eg ID NUM REAL have *semantic values* attached to them, eg ID(n14), NUM(515)
- Reserved words: Punctuation tokens e.g. IF, VOID, RETURN constructed from alphanumeric characters, cannot be used as identifiers

9 February, 2009

IN2009 Language Processors - Session 2

7

## Example informal specification

### Identifiers in C or Java:

- “An identifier is a sequence of letters and digits; the first character must be a letter. The underscore ( `_` ) counts as a letter. Upper- and lowercase letters are different.”
- “If the input stream has been divided into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.

9 February, 2009

IN2009 Language Processors - Session 2

8

## Identifiers in C or Java

- *Blanks, tabs, newlines and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and symbols.”*

*How do we formally specify the description above?*

9 February, 2009

IN2009 Language Processors - Session 2

9

## Formal specifications of tokens

- Approach:
  - Specify lexical tokens using the formal language of regular expressions.
  - Transform regular expressions into deterministic finite automata (DFA)
  - Implement lexical analysers (lexers) using DFA.
  - Fortunately for our sanity, there are automatic conversion tools...

9 February, 2009

IN2009 Language Processors - Session 2

10

## Formal specifications of tokens

*“a formal language is a language that is defined by precise mathematical or machine processable formulas” (Wikipedia)*

- Languages
  - A language is a set of strings
  - A string is a finite sequence of symbols
  - Symbols are taken from a finite alphabet
- Many types of formal languages, at this stage we will consider **regular languages**.

9 February, 2009

IN2009 Language Processors - Session 2

11

## Regular languages

- Regular languages can be:
  - Specified by **regular expressions**
  - Accepted by deterministic/non-deterministic finite state automata (DFA,NFA)
    - A word is accepted by the state machine if and only if the word belongs to the language i.e. it satisfies the regular expression specification
- A regular expression can be translated into an NFA.
- DFAs can be efficiently implemented.
- An NFA can be converted into a DFA.

9 February, 2009

IN2009 Language Processors - Session 2

12

## Regular expressions

- Regular expressions (regex, regexp) - strings that describe the syntax of other strings.
- These are simply *patterns*, constructed by syntactical rules and defining a set of strings.
- Components
  - Alphabet: Valid symbols of the language
  - Operators:  $\cdot$   $|$   $*$   $+$

9 February, 2009

IN2009 Language Processors - Session 2

13

## Regular expressions

- There is a useful connection to real life software development here, too.
  - Regexp can be used in a number of programming languages and tools
  - TextPad, Emacs, etc. allow for searching for text given a regular expression - much more powerful than plain text searching.
  - Java contains a class `RegExp`, too.
- On to the rules...

9 February, 2009

IN2009 Language Processors - Session 2

14

## Regular expressions

### Symbol

- For each symbol  $a$  in the alphabet of the language, the regexp  $a$  denotes the language containing just the string  $a$
- $\text{Lang}(a) = \{ "a" \}$

9 February, 2009

IN2009 Language Processors - Session 2

15

## Regular expressions

### Alternation

- Given 2 regular expressions  $M$  and  $N$  then  $M | N$  is a new regexp.
- A string is in  $\text{lang}(M|N)$  if it is in  $\text{lang}(M)$  or  $\text{lang}(N)$ .
- $\text{lang}(M|N) = \text{lang}(M) \cup \text{lang}(N)$
- E.g.  $\text{lang}(a | b) = \{ "a", "b" \}$  contains the 2 strings  $a$  and  $b$ .

9 February, 2009

IN2009 Language Processors - Session 2

16

## Regular expressions

### Concatenation

- Given 2 regexes  $M$  and  $N$  then  $M \cdot N$  is a new regexp.
- A string is in  $\text{lang}(M \cdot N)$  if it is the concatenation of 2 strings  $\alpha$  and  $\beta$  s.t.  $\alpha$  in  $\text{lang}(M)$  and  $\beta$  in  $\text{lang}(N)$ .
- $\text{lang}(M \cdot N) = \{ \alpha\beta \text{ s.t. } \alpha \text{ in } \text{lang}(M) \text{ and } \beta \text{ in } \text{lang}(N) \}$
- Thus regexp  $(a|b) \cdot a = \{ "aa", "ba" \}$  defines the language containing the 2 strings  $aa$  and  $ba$

9 February, 2009

IN2009 Language Processors - Session 2

17

## Regular expressions

### Epsilon

- The regexp  $\epsilon$  represents the language whose only string is the empty string.
- $\text{Lang}(\epsilon) = \{ "" \}$
- Thus  $(a \cdot b) | \epsilon$  represents the language  $\{ "", "ab" \}$

9 February, 2009

IN2009 Language Processors - Session 2

18

## Regular expressions

### Repetition

- $M^*$  - Kleene closure (or Kleene Star) of  $M$
- A string in  $M^*$  if it is the concatenation of  $\geq 0$  strings, all in  $M$ .
- Thus  $((a|b) \cdot a)^*$  represents the infinite set  $\{ "", "aa", "ba", "aaaa", "baaa", "aaba", "baba", "aaaaaa", \dots \}$

9 February, 2009

IN2009 Language Processors - Session 2

19

## Examples

- $(0|1)^* \cdot 0$
- $b^*(abb^*)^*(a|\epsilon)$
- $(a|b)^*aa(a|b)^*$
- Conventions:
  - omit  $\cdot$  and  $\epsilon$ , assume Kleene closure binds tighter than  $\cdot$  and concatenation binds tighter than  $|$
  - $a \cdot b^*$  means  $a(b^*)$
  - $ab | c$  means  $(a \cdot b)|c$
  - $(a | )$  means  $(a|\epsilon)$

9 February, 2009

IN2009 Language Processors - Session 2

20

## Abbreviations (extensions)

$[abcd]$	means $(a   b   c   d)$
$[b-g]$	means $[bcdefg]$
$^b-g]$ or $\sim[b-g]$	means everything but $[bcdefg]$
$[b-gM-Qkr]$	means $[bcdefgMNO PQkr]$
$M?$	means $(M   \epsilon)$
$M+$	means $M(M)^*$

9 February, 2009

IN2009 Language Processors - Session 2

21

## Regular expression summary

$a$ or $"a"$	ordinary character, stands for itself
$\epsilon$	the empty string
	another way to write the empty string
$M   N$	alternation
$M \cdot N$	concatenation (often written simply as $MN$ )
$M^*$	repetition (zero or more times)
$M+$	repetition (one or more times)
$M?$	Optional, zero or one occurrence of $M$

9 February, 2009

IN2009 Language Processors - Session 2

22

## Regular expression summary

$["a"- "z" "A"- "Z"]$	Character set alternation (JavaCC)
$\sim[]$	Any single character ( $\sim[]$ is JavaCC form)
$"\n" "t" "\\""$	newline, tab, double quote (quoted special characters)
$"a. + *"$	quotation, string stands for itself (in this case $a. + *$ )
$[a-zA-Z]$	Character set alternation
$.$	Any single character except newline

9 February, 2009

IN2009 Language Processors - Session 2

23

## Regular expressions for some tokens

$( " "   "\n"   "\t" )$	<i>no token; whitespace; ignore</i>
$if$	<b>IF</b>
$[a-z][a-zA-Z0-9]^*$	<b>ID</b>
$[0-9]^+$	<b>NUM</b>
$([0-9]^+ \cdot "[0-9]^*" )   ([0-9]^* \cdot "[0-9]^+)$	<b>REAL</b>
$("--"[a-z]^* "\n")$	<i>comment starting --; ignore</i>
$.$	<b>Error</b>

**Completeness:** there must always be a match for some initial substring of the input.

We always want the **longest match**.

**Rule Priority:** If more than one regexp matches the string, pick the first one from the lexical spec.

9 February, 2009

IN2009 Language Processors - Session 2

24

## Finite Automata

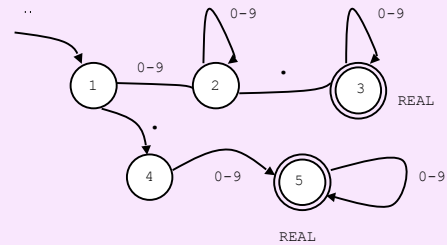
- A finite set of states and edges.
- Edges lead from one state to another. Each edge is labeled by a symbol.
- A single start state and one or more final states.
- Each final state identifies a TOKEN from the language
- A string is accepted if, after all characters have been matched by a transition, the last state is a final state.
- Deterministic or Non-deterministic (NFA) .

9 February, 2009

IN2009 Language Processors - Session 2

25

## Finite automaton



(From Appel Figure 2.3)

9 February, 2009

IN2009 Language Processors - Session 2

26

## DFA Implementation

```

method Token lex() { //example automaton for REAL only
    int state = 1; String text = ""; char ch;
    while (true) {
        ch = nextchar(); // Get the next input character
        text = text + ch; // Collect text of the token
        if (state == 1) {
            if (ch >= '0' && ch <= '9') {
                state = 2;
            } else if (ch == '.') {
                state = 4;
            } else {
                lexerror(ch);
            }
        }
        ...
    }
}

```

9 February, 2009

IN2009 Language Processors - Session 2

27

## DFA Implementation

```

...
else if (state == 5) {
    if ch >= '0' && ch <= '9') {
        state = 5;
    } else {
        return new Token(REAL, new Double(text));
    }
} else {
    error ("Illegal state: shouldn't happen");
}
}

```

- Inefficient and verbose..use transition matrix instead

9 February, 2009

IN2009 Language Processors - Session 2

28

## Efficient DFA implementation

- Create a table that represents the transition matrix
  - `int edges[NumStates][NumCharacters]`
  - `Edges[s][c] = sn`  
If there exists a transition labeled with character `c` that joins states `s` and `sn`
  - Index `c` can be extracted from the ASCII code
  - Create a "dead" state that loops to itself on all characters (to denote no match)
- Create an array that marks states as final and maps them to tokens.
- This table can be generated automatically

9 February, 2009

IN2009 Language Processors - Session 2

29

## From RegExp to Implmentation

- A RegExp can be converted into an NFA.
- However, the implementation of NFAs is inefficient.
- But NFAs can be translated into DFAs!

**All these steps can be automated**  
**There are tools that generate lexical analysers**  
**from lexical specifications**  
**E.g. JavaCC**

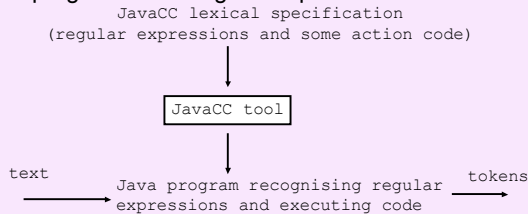
9 February, 2009

IN2009 Language Processors - Session 2

30

## JavaCC compiler-compiler

- Fortunately, tools can produce finite automata programs from regular expressions...



9 February, 2009

IN2009 Language Processors - Session 2

31

## JavaCC token regexp

- Characters and strings must be quoted, eg:
  - “;” “int” “while” “\n” “\hello”
- Character lists [...] provide a shorthand for |, eg:
  - [“a”-“z”] matches “a” through “z”, [“a”, “e”, “i”, “o”, “u”] matches any single vowel, ~[“a”, “e”, “i”, “o”, “u”] any non-vowel, ~[] any character
- Repetition with + and \*, eg:
  - [“a”-“z”, “A”-“Z”]+ matches one or more letters
  - [“a”-“z”][“0”-“9”]\* matches a letter followed by zero or more digits

9 February, 2009

IN2009 Language Processors - Session 2

32

## JavaCC token regexp

- Shorthand with ? provides for optional expr, eg:
  - (“+”|-)?[“0”-“9”]+ matches signed and unsigned integers
- Tokens can be named
  - TOKEN : {  
  < IDENTIFIER: <LETTER> <LETTER> <DIGIT>+ >  
}
  - TOKEN : {  
  < LETTER: [“a”-“z”, “A”-“Z”] >  
  | < DIGIT: [“0”-“9”] >  
}
  - now <IDENTIFIER> can be used later in defining syntax

9 February, 2009

IN2009 Language Processors - Session 2

33

## JavaCC lexical analysis example

```

/* file MyParser.jj */

PARSER_BEGIN(MyParser)
class MyParser {
PARSER_END(MyParser)

TOKEN : {
  < IF: "if" >
  | < #DIGIT: [“0”-“9”] >
  | < ID: [“a”-“z”] ([“a”-“z”] | <DIGIT>)* >
  | < NUM: (<DIGIT>)+ >
  | < REAL: ( (<DIGIT>)+.“(<DIGIT>)* ) |
    ( (<DIGIT>)*.“(<DIGIT>)+ ) >
}

SKIP : { // skipped during lexical analysis
  < “-” ([“a”-“z”])* (“\n” | “\r” | “\r\n”) >
  | “ ” | “\t” | “\n” | “\r”
}
  
```

# means can use **only** in  
TOKEN definitions (local)

9 February, 2009

IN2009 Language Processors - Session 2

34

## JavaCC lexical analysis example

```

void Start() : // the grammar
{Token t;}
{ ( ( t=<IF> | t=<ID> | t=<NUM> | t=<REAL> )
  { System.out.println("token found: "
    + tokenImage[t.kind]
    + "("+t.image+")"); }

  ) * <EOF>
} /* end of file MyParser.jj */

/* file Main.java */
class Main {
  public static void main(String args[]) throws
    ParseException {
    ...
    MyParser parser = new MyParser(System.in);
    parser.Start();
  }
}
  
```

9 February, 2009

IN2009 Language Processors - Session 2

35

## JavaCC introduction

- Generates a combined lexical analyser and parser (a Java class)...here the class is called MyParser
- This session we're learning about lexical analysis
  - JavaCC does this and it is the focus of our first example – we'll see more about complete JavaCC-generated parsers later.
- You can see another example in today's lab.

9 February, 2009

IN2009 Language Processors - Session 2

36

## JavaCC introduction

- all you need to know now is that it also generates a Java method to recognize the things we've labelled under 'Start()'
- to make it work,
  - create an object  
MyParser parser = new MyParser(inputstream)
  - and then call the method...parser.Start()
- a class Token is also generated
  - field **image** is the string matched for the token
  - field **kind** can be used to index array **tokenImage** to see the token type

9 February, 2009

IN2009 Language Processors - Session 2

37

## What you should do now...

- Re-read chapter 2
  - You really need to understand regular expressions.
  - Understanding Finite Automata will be helpful as well.
  - Don't worry too much about the translation algorithms and other theoretical issues
  - Think about how to represent real numbers with exponents using regular expressions... because:

9 February, 2009

IN2009 Language Processors - Session 2

38

## Coursework 1

### Preamble:

- 30% of total coursework.
- Out this Wednesday.
- Due: Friday 27 February
- Individual or declared pairwork.
- Hand in using Cityspace.
- Guard your work, don't risk plagiarism charges by leaving a USB key with your work around, or "sharing answers" etc.

9 February, 2009

IN2009 Language Processors - Session 2

39

## Assessment 1, part 1

- Use JavaCC **regular expressions** to define precisely *integer literals* and *floating point literals*.
- In this context, 'literal' means the piece of text that appears in a program to denote a number (for example, the text '3.142' denotes the number 3.142).
- Implement and test your expressions using JavaCC (make your expressions readable and understandable).

9 February, 2009

IN2009 Language Processors - Session 2

40

## Integer Literals

- An integer literal may be expressed as:
  - *binary*,
  - *decimal*,
  - *hexadecimal*, or
  - *octal* numerals.
- Each may optionally be suffixed with the character **L** to denote an integer of type *long*, and may be prefixed with a **+** or a **-** character to indicate sign.

9 February, 2009

IN2009 Language Processors - Session 2

41

## Integer Literals

- A **decimal numeral** is either the single character **0**, or consists of a digit from 1 to 9, optionally followed by one or more digits from 0 to 9.
- A **binary numeral** consists of the leading characters **0b** or **0B** followed by one or more of the digits **0** or **1**.

9 February, 2009

IN2009 Language Processors - Session 2

42

## Integer Literals

- A **hexadecimal numeral** consists of leading characters `0x` or `0X` followed by one or more hexadecimal digits.
- A *hexadecimal digit* is a digit from 0 to 9 or a letter from `a` through `f` or `A` through `F`.
- An **octal numeral** consists of a digit 0 followed by one or more of the digits 0 to 7.

9 February, 2009

IN2009 Language Processors - Session 2

43

## Integer Literals

- Examples of integer literals:

```
0
19960372
0xDadaCafe
0L
0777L
0xC0B0L
0x00FF00FF
0b00100110
0b110010L
426355690003133711121133114641
```

9 February, 2009

IN2009 Language Processors - Session 2

44

## Floating Point Literals

- A floating point literal has the following parts:
  - a *whole-number part*,
  - a *decimal point* (represented by the period character `.`),
  - a *fractional part*,
  - an *exponent*, and
  - a *type suffix*. A type suffix is either the letter `d` (denoting *double* type) or `f` (denoting *float* type).

9 February, 2009

IN2009 Language Processors - Session 2

45

## Floating Point Literals

- The exponent, if present, is indicated by the letter `e` followed by an optionally signed number.
- At least one digit, in either the whole number or the fraction part is required.
- One of the following is also required:
  - a decimal point,
  - an exponent, or
  - a float type suffix

9 February, 2009

IN2009 Language Processors - Session 2

46

## Floating Point Literals

- All other parts are optional.
- Subject to the previous constraints, the the fractional-part and the number in the exponent are sequences of digits from 0 to 9 (i.e. decimal only).
- The whole-number part is a sequence of digits from 0 to 9 and may optionally be prefixed with a `+` or a `-` character to indicate sign.

9 February, 2009

IN2009 Language Processors - Session 2

47

## Floating Point Literals

- Examples:

<code>1e1f</code>	<code>2.f</code>
<code>.3f</code>	<code>0f</code>
<code>3.14f</code>	<code>6.022137e23f</code>
<code>1e1</code>	<code>2.</code>
<code>0.3</code>	<code>0.0</code>
<code>3.14</code>	<code>1e-9d</code>
<code>1e137</code>	<code>-5.56e4263</code>
<code>-42f</code>	

9 February, 2009

IN2009 Language Processors - Session 2

48