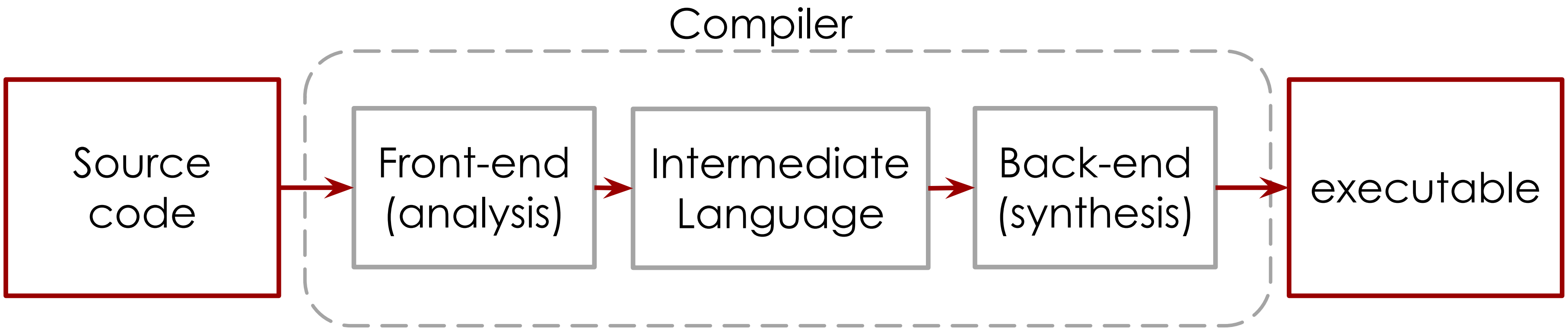
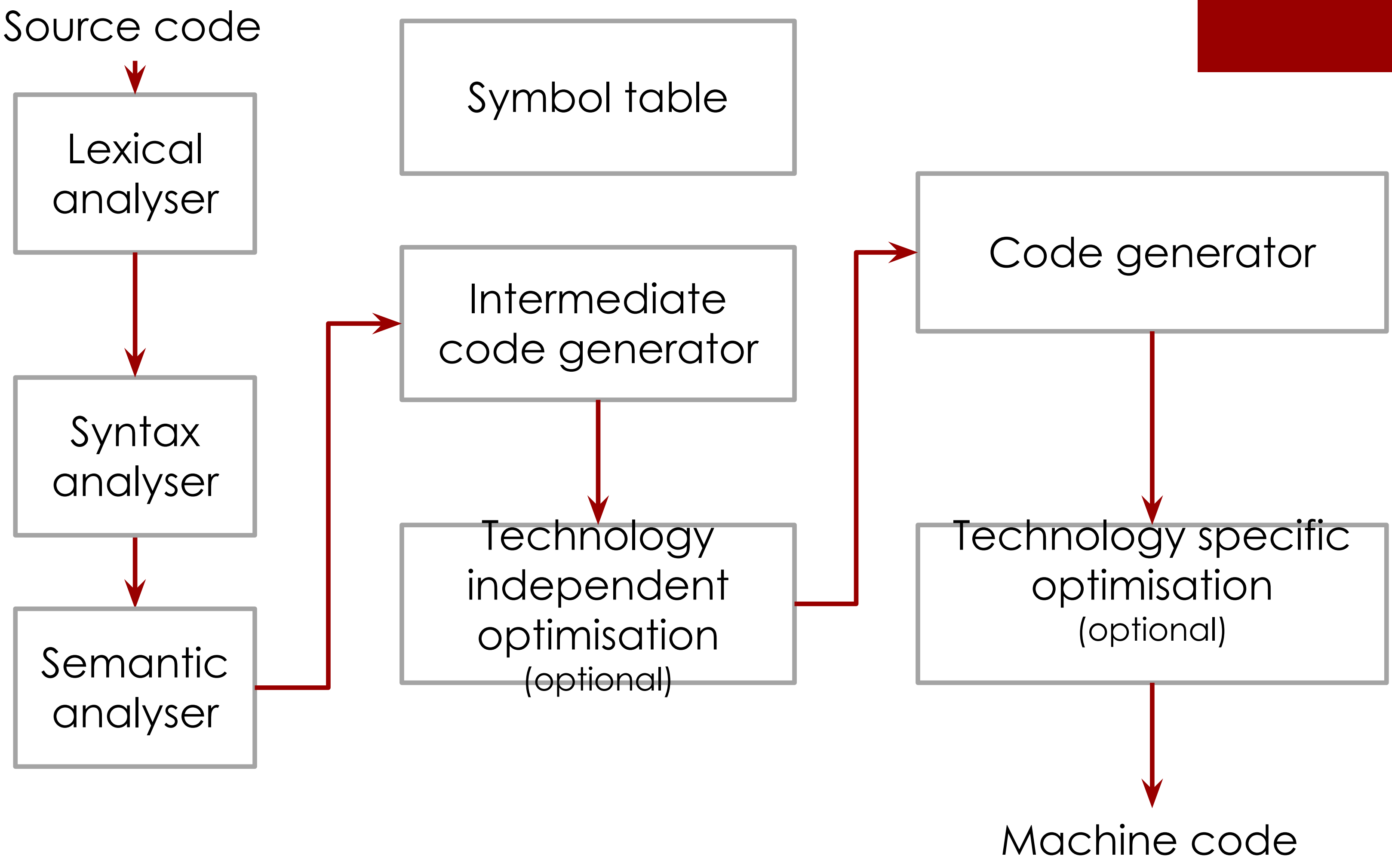
Formal Languages and Compilers

A lexical analyzer generator

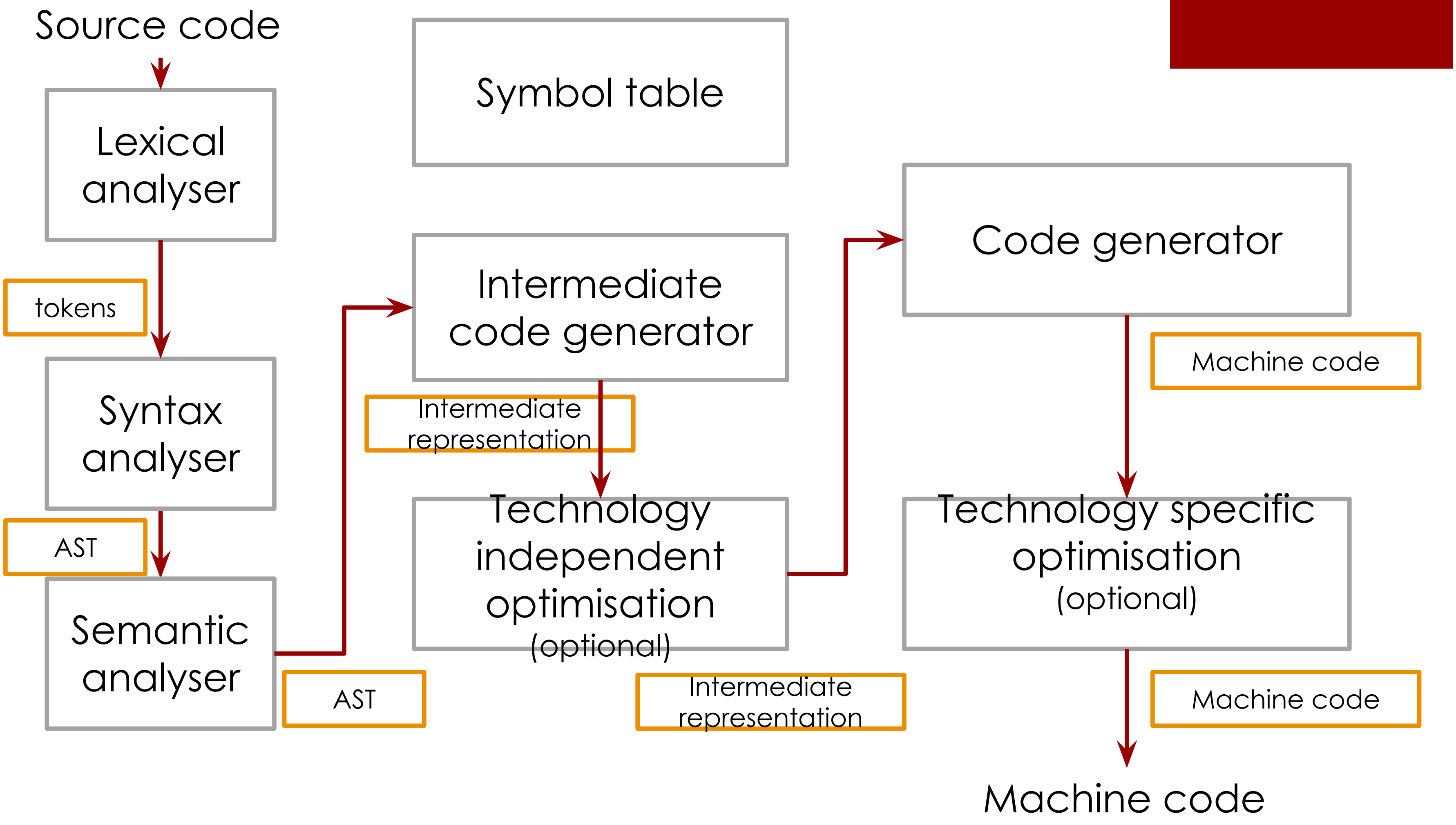
Recap


Compiler
Source code
Front-end (analysis)
Intermediate Language
Back-end (synthesis)
executable

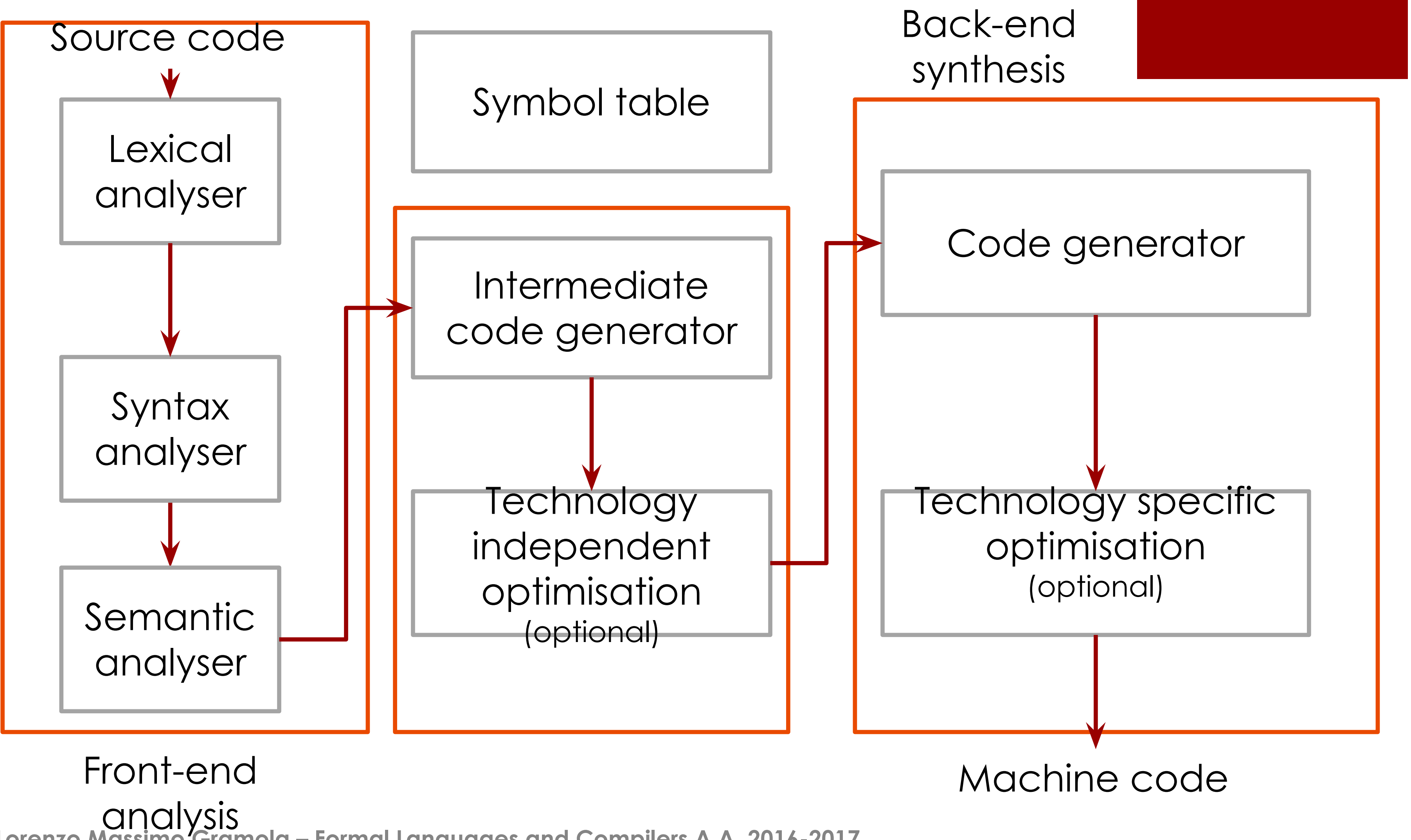
A more detailed view


Source code
Symbol table
Lexical analyser
Syntax analyser
Semantic analyser
Intermediate code generator
Technology independent optimisation (optional)
Code generator
Technology specific optimisation (optional)
Machine code

A more detailed view


Source code
Lexical analyser
tokens
Syntax analyser
AST
Semantic analyser
AST
Symbol table
Intermediate code generator
Intermediate representation
Technology independent optimisation (optional)
Intermediate representation
Code generator
Machine code
Technology specific optimisation (optional)
Machine code
Machine code

A more detailed view



Back to the lab.. so far..

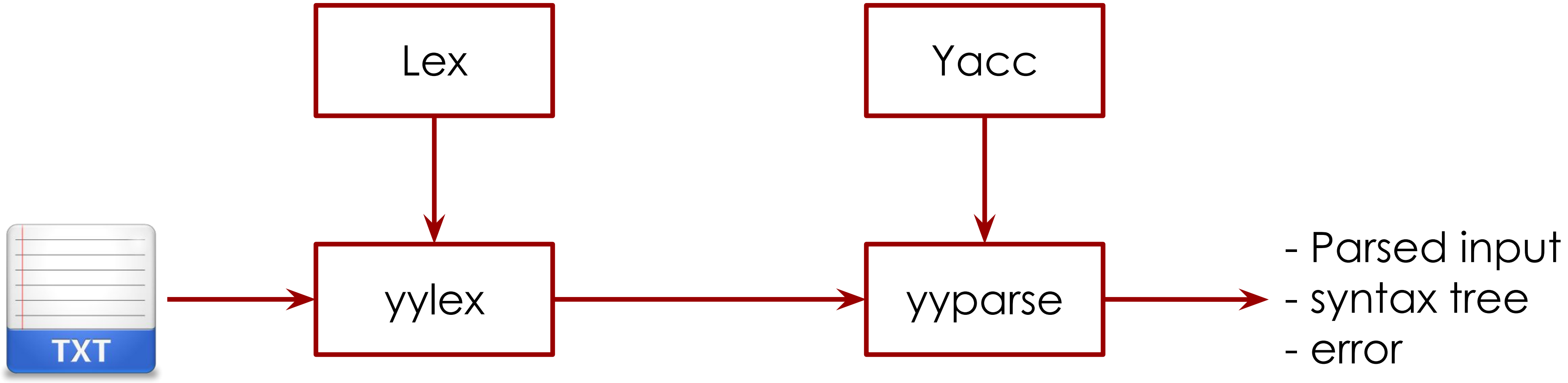
- Lex is used as lexical analyser
- Used to build our front-end
- INPUT: a generic char sequence
- OUTPUT: a series of token

Usage of LEX

- LEX reads the input and performs some ***action***.
- Action are decided based on the regexp matched
- Action can
  - manipulate the input → find a number n, return n+1
  - Omit the output → match something and do nothing
- Lex can be used for different scopes
  - Simple transformation
  - For analysis and statistic gathering (at lexical level)
  - As lexer in the front end part of a compiler/interpreter (our final purpose)

LEX and YACC

- Lex can be used with a parser (YACC) to identify and return TOKENS



Digging in Lex working

- Let’s start knowing Lex a bit more
- Take the file book.csv, parse it and produce an HTML table: every row becomes a row, every value a cell

```
[0-9]+            { ?????       }

","               { ??????      }

\n                {??????       }
```

Digging in Lex working (cont.)

- Form CSV to HTML

```
[0-9]+          {printf("<td> "); ECHO;}

","             {printf("</td> "); }

\n              {printf("</td></tr> \n <tr>"); }
```

- Before starting the parsing(reading) action we give structure to html document

Digging in lex working (cont.)

- MAIN:
  - Default main is provided by lex library
  - Custom main

- Custom main allow us to build some features
  - Performing operation before calling yylex()
  - Chaining multiple files

LEX – what's next?

- LEX reg exp – how to write and use them
- LEX actions – what are actions and what kind of stuff we can do
- LEX functions – user defined functions

LEX regular expressions

- Reg exp are used to define the rules
  {definitions}
  %%
  {rules} ← here we use reg exp + actions
  %%
  {user routines}

- Definitions can be empty and subroutines as well, thus the minimum lex input source is

  %%

  which just copies input to output leaving it unchanged.

LEX reg exp (cont)

- Letters and digit are chars, so they can be easily matched

- ***Operators*** are special chars like

  \[]^-?.*+|()$/{}%<>

  myindex”++” matches exactly *myindex++*

  myindex\+\+ or “myindex++” serves the same scope

LEX reg exp (cont)

- ***Char classes*** specified with the use of []
  defines some class of chars we would like to find

  [abc] → may match a or b or c
  [a-z] → from a to z
  [^abc] →everything except a or b or c

- . → **any char except the new line**

- ***Optional expression***
  ab?c → b can be found or not
  so matches ac or abc as well

LEX reg exp (cont)

- ***Repeated expression***
  a* → Kleene closure
  a+ → positive closure

- So say you find something like
  [a-zA-Z]+
  what does it mean?

- [A-Za-z][A-Za-z0-9]*
  In this case instead what does it mean?
  Does it represent something in your opinion?

LEX reg exp (cont)

- ***Alternation and grouping***
  alternation provided by the pipe |
  a|b → either a or b
  grouping provided by the parenthesis ()
- You can mix all the operators to obtain complex reg exp
- (ab|cd+)?(ef)*
- So what if we want to match any italian car licence plate..? What should we write..?

LEX reg exp (cont)

- ***Context sensitivity***
  LEX can recognize a small amount of surrounding context, for the sake of our purpose we will use the following two simple operators.
- ^ means at the beginning of the line
- $ means at the end of the line (followed by a new line)

Context sensitivity example

- Say you want to apply different set of lexical rules at a different time while reading the input.
- To do so you need to handle the context, for example by using the ^ operator or $ context operator.
  (^ recognizes immediately left preceding context
  $ recognizes immediately right following context)
- You could use *starting conditions* → a rule is associated with a condition, such rule is recognized only when Lex is in a starting condition

Context sensitivity example

- Suppose you want to copy input to output, but changing the word “directory” (when found) to
  - “first” if the line begins with letter a
  - “second” on every line which begin with letter b
  - “third” …. With letter c

  the other occurrences are left unchanged, just for the sake of such example.
- How to do that?

Use starting conditions

- In definition section
  %start cond1 cond2 cond3
  condition name can have any order

- Condition may be referenced at head of a rule
  <condX> expr
  recognized only when Lex is in such condition

- Enter a condition with *BEGIN cond*
  resume lex state with *BEGIN 0*

- At home: do the previous exercise with starting conditions.

LEX Actions

- Bound to one or more rules – executed when the pattern matches
- ***Default action = do nothing. Copy input to output.***
- Action can be meant as what is performed instead of copying input to output.

  EG. You want to look for a given word (a search)
  [givenword]   {printf(“%s\n”, ***yytext***)};

  This action is so common that LEX provides a macro for it: ECHO

LEX Actions (cont)

- An action can contain a C statement/instruction

  EG
  {word}+ {wordcount++; charcount+=***yyleng***;}

  upon finding a "word", you increment the word count and the char count - based on definition of word

- We have seen **yytext** and **yyleng.** The latter is intuitive and contains the length of the matched string, while the first is an array of chars containing the part of the input stream that matched the pattern.

Ambiguous matching

- Some specification may be ambiguous

```
integer         {action 1}
[a-z]+          {action 2}
```

- What does happen if we give as input “integer” ?
- And “integers”?

Handling ambiguous spec

- Longest match wins
- Among rules with the same number of chars… specification order is used
- So in the previous example on “integer” input, the action 1 is preferred over rule corresponding to action 2.

Exercise: count instances of a word

- Say you want to count instances of *he* and *she* in a text.
- How do you proceed?

Exercise: count instances of a word

- Say you want to count instances of *he* and *she* in a text.
- A possible pattern matching can be

```
she     s++;
he      h++;
\n       |
.    ;
```

- Does it work?

Exercise: count instances of a word

- Say you want to count instances of *he* and *she* in a text.

- A possible pattern matching can be

```
she        s++;
he         h++;
\n          |
.     ;
```

- No → a matched *she* has consumed the *he* as well.

Exercise: count instances of a word

- Say you want to count instances of *he* and *she* in a text.

- Solution is to use **REJECT**.

```
she          {s++; REJECT;}
he           {h++; REJECT;}
\n            |
.       ;
```

Lex – definition section

- Recall the format of the lex source file: definitions for lex are those given before the first %%

```
{definitions}
%%
{rules}
%%
{user routines}
```

- Used to define additional options and/or variables to be used in the program or by Lex
- Everything not interpreted by Lex is taken "as is" and placed in the output program
- 3 kind of stuff:
  - code in %{ }%
  - everything after the last %%
  - anything which is not part of rules or actions and which begins with a blank or tab (eg. comments)

Compile and run

- 2 steps
- 1) Lex → generated program in the host general purpose language
- 2) compile and load such program with appropriate (and needed)

```
lex –o somefile.c file.l                (lex –t file.l > somefile.c)
(g)cc somefile.c –o somefile.o –c
(g)cc somefile.o –o myprogram –ll
```

- As you may have guessed Lex and Yacc goes hand in hand. Lex produces tokens and Yacc organise the grammar.

Excercises

- For the next time
- 1) Build a source file for lex such that upon reading a number is going to output the same number + 3 if the number can be divided by 7, else leave the output unchanged.
- 2)Write a lexer that takes in input more files

Bibliography

- Compilers 2nd edition – Aho, Lam, Sethi, Ullman
- Lex − A Lexical Analyzer Generator *M. E. Lesk and E. Schmidt*