Regular Languages and Representation in the Real World

5DV037 — Fundamentals of Computer Science

Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Three Basic Areas of Application

- Three basic areas in which regular expressions are used in applications within computer science are the following.
  - Lexical analysis in the compilation of programming languages and the processing of natural (human) language.
  - (Extended) regular expressions in programming environments.
  - Modelling computer systems using finite-state machines.
- A very brief overview of these areas will be presented.

Motivation for Lexical Analysis

- Consider the tabloid headline:

Man bites dog!

- Rather than processing each character separately, one naturally breaks it into three words and a terminating punctuation mark.
- The need for this is more apparent if the sentence is in a language which is unfamiliar.

Homme mord chien !

- To understand the sentence, one might strip away the punctuation marks and look up each word separately in dictionary and then try to piece together the results.
- The key is that the input is first broken into words.
- In processing a program in a computer programming language, a similar process is involved.

Lexical Analysis for Computer Programs

- Consider the following program fragment from a fictitious language:

  ```
  While X1<=X2*1.25E12 do X2:=foo(X1); end while;
  ```

- The first thing that a compiler will do is to break up this string into "words" and "punctuation marks", called *tokens*.
- This string has the following 17 tokens (separated by spaces):

  ```
  While X1 <= X2 * 1.25E12  do X2 := foo ( X1 ) ; end while ;
  ```

- These tokens include:

  keywords: `While do end while`

  identifiers: `X1 X2 foo`

  numbers: 1.25E12

  operators: `<= :=`

  punctuation: `( ) ;`

- Just as words in a natural language, tokens have meaning only as units.
- <=, <, and = are each distinct tokens with distinct meanings, just as "do", "or", and "door" are distinct words in English.

Lexical Analysis for Computer Programs 2

- The language which describes tokens is almost always regular.
- There are tools to build an accepter automatically from the description of a regular language.
  - Lex, Flex, SimpLex
- These classical tools were built to generate C code, but variations for other host languages have been developed as well.

Clarification: C is the language in which the compiler is written, not the language which is to be compiled.

- These tools take a representation (using REs or regular grammars) and produce an NFA which accepts the language consisting of legal tokens.
- Instead of just answering yes or no, there is a program associated with each accepting state. It is executed when the NFA halts in that state.
- Nondeterminism is handled by ordering the accepting states, and running the program for the first one in the list which accepts.

A Simple Example of Token Description

- Here is a simple recursive description, using regular expressions, of the numbers in a typical programming language.
- First some special names to avoid symbol conflicts:

$$\langle \text{plus} \rangle = \text{the symbol } + \qquad \langle \text{dot} \rangle = \text{the symbol } .$$
$$\langle \text{minus} \rangle = \text{the symbol } -$$

- Now the main definitions:

$$\begin{aligned}
\langle \textit{digit} \rangle &= 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 \\
\langle \textit{sign} \rangle &= \langle \textit{plus} \rangle + \langle \textit{minus} \rangle + \lambda \\
\langle \textit{integer} \rangle &= \langle \textit{digit} \rangle \cdot \langle \textit{digit} \rangle^* \\
\langle \textit{signed_int} \rangle &= \langle \textit{sign} \rangle \cdot \langle \textit{integer} \rangle \\
\langle \textit{dec_num} \rangle &= \langle \textit{signed_int} \rangle \cdot \langle \textit{dot} \rangle \cdot \langle \textit{integer} \rangle \\
&\quad + \langle \textit{sign} \rangle \cdot \langle \textit{dot} \rangle \cdot \langle \textit{integer} \rangle + \langle \textit{signed_int} \rangle \\
\langle \textit{exp_num} \rangle &= \langle \textit{dec_num} \rangle + \langle \textit{dec_num} \rangle \cdot \mathrm{E} \cdot \langle \textit{signed_int} \rangle \\
\langle \textit{real_num} \rangle &= \langle \textit{dec_num} \rangle + \langle \textit{exp_num} \rangle
\end{aligned}$$

- Modulo some syntactic conventions, this sort of specification is actually used in the specification of lexical analyzers.

Alternate Represenatation of Token Descriptions

Note: It is more common to express such defintions using a grammar formalism, although the two representations differ only in minor ways.

$$
\begin{aligned}
\langle digit \rangle &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle sign \rangle &\rightarrow \langle plus \rangle \mid \langle minus \rangle \mid \lambda \\
\langle integer \rangle &\rightarrow \langle digit \rangle \cdot \langle digit \rangle^* \\
\langle signed\_int \rangle &\rightarrow \langle sign \rangle \cdot \langle integer \rangle \\
\langle dec\_num \rangle &\rightarrow \langle signed\_int \rangle \cdot \langle dot \rangle \cdot \langle integer \rangle \\
&\quad \mid \langle sign \rangle \cdot \langle dot \rangle \cdot \langle integer \rangle \mid \langle signed\_int \rangle \\
\langle exp\_num \rangle &\rightarrow \langle dec\_num \rangle \mid \langle dec\_num \rangle \cdot \mathrm{E} \cdot \langle signed\_int \rangle \\
\langle real\_num \rangle &\rightarrow \langle dec\_num \rangle \mid \langle exp\_num \rangle
\end{aligned}
$$

Regular Expressions in Programming

- An extended form of regular expressions (often called *regexp*s is widely used in programming and systems applications (*e.g.*, *GnuEmacs*).
- There are two principal flavors, which differ in minor ways:
  - The *POSIX* regexps, used in Unix/Linux and their friends.
  - The *Perl* flavor.
- Compared to the regular expressions which have been studied in this course, these differ in two principal ways.
  - They allow for the recall of matched patterns, so they are strictly more powerful than the REs in the theory world.
  - They have syntactic conventions adapted to the real world:
    - Expressions may include symbols which are reserved for punctuation in the formal REs, such as "+", "(", and the like.
    - Ways to match special characters, such as tabs and the beginning or end of a line, are included.
    - Abbreviations to match common sets, such as all letters, or all uppercase letters, are available.

Some Syntax for Regexps

- The real world regexps have a very ugly syntax which is difficult to read, but it is not clear that there are better alternatives.
- Some examples:

Classical REs	Posix	Perl	Comments
a+b	a|b	a|b	
$a \cdot a^*$	a+	a+	at least one occurrence
$a + \lambda$	a?	a?	zero or one occurrence
complex	a{m,n}	a{m,n}	$\geq$ m and $\leq$ *n* occurrences
(a+b+c)	[abc]	[abc]	
(e)	\(e\)	(e)	grouping
b+c+...+g	[b-g]	[b-g]	Any range of letters or digits
complex	[^b-g]	[b-g]	Anything except b through g
complex	.	.	any character
	\n	\n	linefeed
	^	^	beginning of line
	$	$	end of line

Some Syntax for Regexps 2

- Special characters are generally escaped to obtain the literal character:
  - `\[` gives the bracket character `[`; likewise for `\]` and `]`.
  - Note that the left parenthesis (is ordinary in Posix but special in Perl. Likewise for) and |.

Example: Match a legal e-mail address (simplified):

```
\b[A-Za-z0-9_%+-]+@[A-Za-z0-9-]+(\.[A-Za-z]+){1,4}\b
```

- `\b` matches the empty string.

Matching in Regexps

- The extended language also allows copying of previously matched expressions.
- The `sed` stream editor uses Posix regexps for editing scripts.
- This little script uses the `sed` stream editor to translate a semicolon-separated list into an SQL insert statement.

```
#!/bin/sh
sed "s/\(.*\)\;\(.*\)\;\(.*\)/Insert into Student Values('\1','\2','\3'\)\;/"
```

- Sed format: `s/`⟨*regexp*⟩`/`⟨*result*⟩`/`
- `\i` matches the $\texttt{i}^{th}$ pattern, for $1 \leq i \leq 9$.

```
wellensittich[17]===>cat testdat.sxc
Aardvark, Alvin A;1234456-7890;aardvark
Perfect, Penelope P;555555-5555;penny
Zebra, Zelda Z;987654-3210;zebra
```

```
wellensittich[18]===>./sedscript.sh < testdat.sxc
Insert into Student Values('Aardvark, Alvin A','1234456-7890','aardvark');
Insert into Student Values('Perfect, Penelope P','555555-5555','penny');
Insert into Student Values('Zebra, Zelda Z','987654-3210','zebra');
```

- This match-and-retrieve feature gives accepting power beyond REs.

Modelling Computer Systems Using DFAs

- Finite automata are often used to model the behavior of certain aspects of a computer system.
- The issue is generally not so much acceptance as the characterization of some property based upon the current state of the machine.
- You will see many examples during your studies of computer science.