Language processors F29PL2, Lecture 1

Jamie Gabbay

February 2, 2014

About me

My name is Murdoch James Gabbay—everybody calls me Jamie. I got a PhD in Cambridge in 2001. Since then I have been a professional researcher—that is, I make my living by discovering and proving mathematical certainties.

My job consists of 50% academic research (into logic and lambda-calculus), 50% teaching (F28PL, F28FS, 29LP), and 50% administration.

Lectures are Mondays at 14:15 in EM3.36, and Tuesdays at 10:15 in EM2.44 and 14:15 in EM1.82.

The course webpage is `http://www.macs.hw.ac.uk/~gabbay/F29LP/`. You need to keep an eye on it.

About you

You are about twenty years old, studying F29LP2 (Language Processors). You are in your third year at university.

You need to turn up to all lectures, do all exercises, and understand material from the one lecture before the next. Anything less is making trouble for yourselves and for your colleagues.

You also need to ask questions. If you do not understand something, ask a question about it during the lecture. This helps me to help you because I get a feeling for what to explain and what to whip through quickly; it also helps your colleagues.

Students who sit passively during lectures, do not make independent efforts to understand the material, and generally 'let it slide'—tend to do poorly.

About the course

I will teach the first half of the course; my colleague http://www.macs.hw.ac.uk/ gg112/ will teach the second half.

Topics for the first half are listed in the Module Descriptor as follows:

About the course

- Lexicon: non-terminal & terminal symbols, type 3 grammars, regular expressions, finite state automata, Moore & Mealey machines, left & right recursion.
- Concrete syntax: type 2 grammars & BNF, factoring grammars, converting left to right recursion, LL(K) & LR(K) grammars, push down automata, parsing.
- Abstract syntax from concrete syntax.
- Static semantics: types.
- Dynamic semantics: overviews of axiomatic, denotational & operational semantics; introduction to structural operational semantics (SOS) for declarations, expressions, and statements

That's quite a lot! So let's get to it.

Formal languages

For this course, 'language' = formal language: the code that programmers use to interact with computers. Obvious examples are: C, ML, and Pascal.

However, there are many simpler examples of formal languages, such as:

- 'the set of dates of the form DD/MM/YY',
- 'your full first and last names and middle initial',
- 'ASCII strings that match `[a-zA-Z0-9]*` (the so-called 'alphanumeric strings'),
- 'well-formatted HTML',
- 'predicates of first-order logic' (like $\forall x.(x{=}y)$),
- 'arithmetic expressions' (like $1 + 1$ or $1 + x$).

Examples of formal languages

Formal languages predate computers as you know them.

Logical languages like propositional or first-order logic are formal languages.

Without you knowing it, the secondary school algebra problems like "$3x + 5 = 7$; find x" are also written in a formal language (of arithmetical algebra).

When you are asked to write a program to verify that (say) a credit card number input by a user in some field in a webpage has the correct format—you're writing a program to verify that a certain string is in the formal language of 'correctly formatted credit card numbers'.

Formal languages are everywhere.

Tokens

A formal language has tokens. Examples of tokens:

- 0, 1, 2, 3, ...
- $+$, $-$, $\div$, $\times$, ...
- $\forall$, $\exists$, $\int$, $\prod$, $\mathcal{И}$.
- $\bot$, $\top$, $\Rightarrow$, $\wedge$, $\vee$, $\neg$, ...
- a, b, c, ..., A, B, C, ...
- α, β, γ, ...,
- $\aleph$, $\beth$, $\gimel$, ...,
- $\mathfrak{A}$, $\mathfrak{B}$, $\mathfrak{C}$, ...,
- (I'd do Russian, Thai, Japanese, and Chinese if this was convenient. Please imagine them inserted here.)

Unicode is an attempt to comprehensively collect all tokens ever made. Bravo.

Tokens

What you consider a 'token' depends to some extent on your desired level of abstraction:

- From the point of view of C, 'then' and 'else' might be considered to be atomic tokens. (Anybody remember the ZX Spectrum, by any chance?)
- From the point of view of the file management system they are token strings (with four tokens each).
- From the point of view of the hard drive platters, each individual character is a token string, where the tokens are 0 or 1 and are written magnetically on a hard drive somewhere.

So do not be alarmed if I call 'then' a token in one lecture, and a token string (sentence) in another lecture.

I'm just messing with our levels of abstraction.

The ZX Spectrum: a token for each keyword

The most basic definition

A formal language consists of:

- A set of atomic symbols called tokens.
- A set of strings of these tokens, which we call sentences.

That's it!

Regular expressions

Of all the things in this course, regular expressions (or regexps) seem to me to be most likely to be useful. Were it up to me, I would put regexps in the school curriculum.

Regexps are a way to specify formal languages, where tokens are (generally) taken to be ASCII characters.

A regexp determines the language which is the set of strings that match that regexp. Examples of regexps:

Regular expressions

The regexp `a*` specifies the language that is the set of possibly empty strings of as:

$$\text{language}(\texttt{a*}) = \{[],\ \texttt{a},\ \texttt{aa},\ \texttt{aaa}, \ldots\}$$

`abc$` specifies the language that is the set of strings ending in `abc`:

$$\text{language}(\texttt{abc\$}) = \{\texttt{abc},\ \texttt{aabc},\ \texttt{babc},\ \ldots \texttt{8abc},\ \ldots \texttt{dbd343Eabc},\ \ldots\}$$

`Hello world\b` specifies 'Hello world' followed by a word boundery:

$$\text{language}(\texttt{Hello world\textbackslash b}) = \{\texttt{Hello world!}, \ldots\} \not\ni \texttt{Hello worlds}$$

Regular expressions

Regexps use the following special syntax:

- `.` Matches any single character except for newline. `\n` represents newline. `\t` represents tab.
- R^* Matches zero or more occurrences of R.
- `[]` Matches any character in the brackets.
  If the first character is a circumflex `^` then it matches any charater not in the brackets.
  Dash indicates a character range, *e.g.* `[0-9]` is equivalent to `[0123456789]` and `[a-zA-Z]` is equivalent to . . . well you write it out.

More regular expressions!

- `^` Matches the beginning of a line. (Also used as negation within square brackets.)
- `$` Matches the end of a line.
- `{}` Indicates how many times the previous pattern can match when containing one or two numbers.
  `Z{1,3}` matches one to three occurrences of the letter `Z`.
- `\` Escapes meta-characters, *e.g.* `\*` denotes asterisk.

Even more regular expressions

- R^+ Matches one or more occurrences of R, *e.g.* `[0-9]+` matches `1`, `131`, *etc*, but not the empty string.
- $R?$ Matches zero or one occurrence of R, *e.g.* `-?[0-9]+` matches a signed number including an optional leading minus.
- `|` Disjunctive matches, *e.g.* `horse|pig|sheep` matches with `horse`, `pig` or `sheep`, but not `horsepig`.

More...

`"..."` Interprets everything within the quotation marks literally, *e.g.* `"Z*"`.

R/S Matches R provided it is followed by S. *E.g* `0/1` matches with `0` in the string `01` but would not match with the string `02`.

`()` Groups a series of regular expressions together into a new regular expression, *e.g.* `(01)` represents the character sequence `01`. Parentheses are useful when building up complex patterns with `*`, `+` and `|`.