APT Session 6: Regular expressions and parsing

Laurence
Tratt

KING'S College LONDON

Software Development Team
2015-02-11

What to expect from this session

1. Using regular expressions in practise.
2. Building a parser.

Prerequisites

1. Have the programming language of your choice (e.g. Java, Python) installed and running on your computer.

Regular expressions: motivation

- A *Domain Specific Language (DSL)* for flexibly matching against text.
- Lots of minor variants, but the basic concepts shared by all implementations.
- Packs very complex matching down to a few characters.

Regular expressions: motivation

- A *Domain Specific Language (DSL)* for flexibly matching against text.
- Lots of minor variants, but the basic concepts shared by all implementations.
- Packs very complex matching down to a few characters.
- Terseness a pro and a con.

Regular expressions: motivation

- A *Domain Specific Language (DSL)* for flexibly matching against text.
- Lots of minor variants, but the basic concepts shared by all implementations.
- Packs very complex matching down to a few characters.
- Terseness a pro and a con. *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.* – Jamie Zawinski, 1997

Regular expressions: the basics

- A regular expression is a series of *atoms*.
- Individual letter and number atoms match themselves. i.e. ‘normal’ text matches as per ‘normal’ expectations.

Regular expressions: the basics

- A regular expression is a series of *atoms*.
- Individual letter and number atoms match themselves. i.e. 'normal' text matches as per 'normal' expectations.

Exercises:

1. Download the words file.
2. Write a program which loads the file in and prints out every line which contains the string 'aab' using normal string search.
3. Modify your program to do the same search, but using a regular expression. You will probably need to *compile* the regular expression to a *Pattern*; then use that compiled pattern to *match* against your string of interest and produce a *Match* which will tell you if the search succeeded or not.

Regular expressions: variable matching

- A dot `.` matches against any character.
- A `*` matches against 0 or more of the preceding atom. A `+` matches against 1 or more of the preceding atom. A *?* matches against 0 or 1 of the preceding atom.
- A character class `[abc]` matches against any of a, b, c. An initial `^` negates this: `[^abc]` matches any character except a, b, c.
- Character sets can contain ranges. `[0-3]` is equivalent to `[0123]`; similarly, `[a-d]` is equivalent to `[abcd]`.

Regular expressions: variable matching

- A dot `.` matches against any character.
- A `*` matches against 0 or more of the preceding atom. A `+` matches against 1 or more of the preceding atom. A *?* matches against 0 or 1 of the preceding atom.
- A character class `[abc]` matches against any of a, b, c. An initial `^` negates this: `[^abc]` matches any character except a, b, c.
- Character sets can contain ranges. `[0-3]` is equivalent to `[0123]`; similarly, `[a-d]` is equivalent to `[abcd]`.

Exercises:

1. Print out all lines which have a ‘z’, followed by one of ‘f’ or ‘g’.
2. Print out all lines which have a ‘z’, followed by any character except ‘f’ or ‘g’.
3. Print out all lines which have a capital letter followed later by a lower-case ‘z’.

Regular expressions: anchoring

- `^` (caret) only matches at the beginning of a line.
- `$` only matches at the end of a line.

Regular expressions: anchoring

- `^` (caret) only matches at the beginning of a line.
- `$` only matches at the end of a line.

Exercises:

1. Print out all lines which have a ‘z’ at the beginning and a ‘z’ somewhere else in the word.
2. Print out all lines which have a ‘z’ at the beginning and end with a ‘z’.

Regular expressions: groups

- Atoms can be grouped with brackets `()` to form a new atom.
- Brackets create groups, which can be referred to later, and sub-parts of text extracted.

Regular expressions: groups

- Atoms can be grouped with brackets `()` to form a new atom.
- Brackets create groups, which can be referred to later, and sub-parts of text extracted.

Exercises:

1. Find every word which contains a ‘z’, then one non-‘z’ character, and then another ‘z’. Print out only those 3 characters for each match.
2. Find every word which contains a ‘z’ followed by one other arbitrary character, and a second ‘z’ followed by one other arbitrary character. Print out the 4 characters matched. Note: the 4 characters do not have to be consecutive.

Regular expressions: greediness

- `*` and friends are *greedy*: they matches as many characters as possible. Greedy matching is often dangerous; it's generally better to explicitly use non-greedy.
- Adding `?` as a prefix to repetition operators (e.g. `*?`) uses non-greedy matching.

Regular expressions: greediness

- `*` and friends are *greedy*: they matches as many characters as possible. Greedy matching is often dangerous; it's generally better to explicitly use non-greedy.
- Adding `?` as a prefix to repetition operators (e.g. `*?`) uses non-greedy matching.

Exercises:

1. Create a new file with the contents `<b>bold</b>`.
2. Find all matches of b followed by another character; print out only those 2 characters.
3. Print all HTML tags in the input file, each on a new line.

Regular expressions: other useful bits

- You can use regular expressions to split apart strings.
- You can use regular expressions to replace strings.

Regular expressions: other useful bits

- You can use regular expressions to split apart strings.
- You can use regular expressions to replace strings.
- Unix's `grep` allows you to do regular expression searches over files and directories (albeit with a slightly limited variant of regular expressions). I use `grep` heavily when programming with unfamiliar codebases.
- My srep tool allows you to do regular expression search and replaceover files and directories.

Parsing

- Parsing is the act of taking a stream of characters and deducing if and how they conform to an underlying grammar. For example the sentence ‘Bill hits Ben’ conforms to the part of the English grammar rule ‘noun verb noun’.

Parsing

- Parsing is the act of taking a stream of characters and deducing if and how they conform to an underlying grammar. For example the sentence ‘Bill hits Ben’ conforms to the part of the English grammar rule ‘noun verb noun’.
- Parsing is the first step in a compiler, analysing the user’s text input and turning it into a tree to make later analysis possible.
- Simplest way of doing so: first *tokenize* the text (i.e. split it into separate words, removing whitespace); second *parse* into a tree.

Parsing

- Parsing is the act of taking a stream of characters and deducing if and how they conform to an underlying grammar. For example the sentence ‘Bill hits Ben’ conforms to the part of the English grammar rule ‘noun verb noun’.
- Parsing is the first step in a compiler, analysing the user’s text input and turning it into a tree to make later analysis possible.
- Simplest way of doing so: first *tokenize* the text (i.e. split it into separate words, removing whitespace); second *parse* into a tree.
- There are *many* ways of doing parsing. Earley, GLR, LL, LR, PEG etc.
- We’re going to start with the simplest: a recursive descent parser. You can use this technique easily in any programming language.

Tokenization

- A token is a *(type, value)* pair.
- The input `a = 0;` has 4 tokens: *(ID, `a`)*, *(=, `=`)*, *(INT, `0`)*, *(;, `;`)*. All whitespace is munched.

Tokenization

- A token is a *(type, value)* pair.
- The input `a = 0;` has 4 tokens: *(ID, `a`)*, *(=, `=`)*, *(INT, `0`)*, *(;, `;`)*. All whitespace is munched.

Exercises:

1. Create a `Token` class which has two fields: `type` and `value`.
2. Using regular expressions, tokenize the input `a = 0;` as per the example above. Your tokenizer should be a loop which starts at position 0 in a string and which tries all token matches until one is successful, at which point the next position is updated.

BNF grammar and recogniser

- We write computer grammars in BNF form: `R ::= S1 S2 ... Sn` where `R` is a rule name and S_n is a symbol. Symbols either reference tokens (e.g. `ID`) or other rules (e.g. `R`).
- Before we write a 'proper' parser, we can write a recogniser which says whether a string conforms to a grammar or not.
- A rule `R ::= X` (where `X` is a token) becomes a function `parse_R(toks, i)` where `toks` is a list of tokens and `i` the current parsing position in `s`. The function returns `-1` if it could not match; or a new `i'` (which must be $> i$) if successful.

BNF grammar and recogniser

- We write computer grammars in BNF form: `R ::= S1 S2 ... Sn` where `R` is a rule name and S_n is a symbol. Symbols either reference tokens (e.g. `ID`) or other rules (e.g. `R`).
- Before we write a 'proper' parser, we can write a recogniser which says whether a string conforms to a grammar or not.
- A rule `R ::= X` (where `X` is a token) becomes a function `parse_R(toks, i)` where `toks` is a list of tokens and `i` the current parsing position in `s`. The function returns `-1` if it could not match; or a new `i'` (which must be $> i$) if successful.

Exercises:

1. For the grammar `Assign ::= ID = INT ;` write a recogniser.
2. Test your recogniser against the inputs `x = 2;` and `y = ;` and `x = y;` at a minimum.

Referencing rules

- Rules can call other rules. Failure propagates. Recursion is allowed provided at least one token is consumed first.
- *But* rules can specify alternatives. `R ::= A | B` means one can parse `R` as `A` *or* `B`. [Note: in most parsing algorithms, neither `A` or `B` has priority. In recursive descent parsers, `A` is tried before `B`. This has some subtle effects which we'll ignore.]

Referencing rules

- Rules can call other rules. Failure propagates. Recursion is allowed provided at least one token is consumed first.
- *But* rules can specify alternatives. `R ::= A | B` means one can parse `R` as `A` *or* `B`. [Note: in most parsing algorithms, neither `A` or `B` has priority. In recursive descent parsers, `A` is tried before `B`. This has some subtle effects which we'll ignore.]

Exercises:

1 Write a recogniser for:

```
Assign ::= ID = Expr ;
Expr   ::= INT + Expr
         | INT
```

2 Test your recogniser against the inputs `x = 2;` and `y = ;` and `x = y;` and `x = 2 + 3;` and `x = 2 + 3 + 4;` and `x = 2 + ;` at a minimum.

Parse trees

- Recognisers are rarely useful. What we really want is a *parse tree*. e.g. `2 + 3` $\Rightarrow$ *add(int(2), int(3))*.
- Building up a tree as we go along is easy. For each element we want (e.g. *Add*), make a class which can hold its contents. When we parse such a thing, instantiate that element.
- Instead of just returning `i'`, return a pair (*tree*, *i'*).

Parse trees

- Recognisers are rarely useful. What we really want is a *parse tree*. e.g. `2 + 3` ⇒ *add(int(2), int(3))*.
- Building up a tree as we go along is easy. For each element we want (e.g. *Add*), make a class which can hold its contents. When we parse such a thing, instantiate that element.
- Instead of just returning `i'`, return a pair (*tree*, *i'*).

Exercises:

1. Write a parse-tree creating parser for:

```
Assign ::= ID = Expr ;
Expr   ::= INT + Expr
         | INT
```

2. Add a `pp` function to each parse tree element so that it can print itself out and you can see if the parsed tree is consistent with the input.

A realistic grammar

Exercises:

1 Write a parse-tree creating parser for:

```
Stmt   ::= Assign | While
Assign ::= ID = Expr ;
Expr   ::= INT + Expr
         | INT
While  ::= WHILE Expr { stmt* }
```

Post-session exercises

Try these (no particular order):

- Discover how to encode precedences for `+-*/` and friends (hint: look for grammars which have rules with names like `factor` and `term`).
- Use a parsing toolkit like ANTLR.
- Experiment with our Eco editor, which uses grammars extensively.