Latest: HOWTO: Get tenure
Next: Writing an interpreter, CESK-style
Prev: Hunting down my son's killer
Rand: Logical literacy

Parsing regular expressions with recursive descent

[article index] [email me] [@mattmight] [+mattmight] [rss]

Regular expressions are a powerful tool for describing, matching and extracting patterns in text.

Every programmer ought to have the experience of implementing a tool for matching regular expressions from scratch.

This article discusses the first phase in this process: parsing core regular expressions using a hand-written recursive descent parser.

One could then proceed with a conventional implementation using conversion to nondeterministic finite automata and then deterministic finite automata. Alternatively, one could match regular expressions with derivatives.

Regular expressions

As commonly used, regular expressions are a pattern language for describing, extracting and manipulating regions of text.

I've discussed regular expressions and their application before and there are plenty of plenty of good books on the topic.

A grammar for regular expressions

In order to match against a regular expression, we first need to parse it.

Parsing reveals the grammatical structure of the regular expression, which is essential for computing the derivative.

In order to parse, we need a context-free grammar for regular expressions.

Luckily, it's easy to construct a simple EBNF grammar for them:

```
<regex> ::= <term> '|' <regex>
          | <term>

<term> ::= { <factor> }

<factor> ::= <base> { '*' }

<base> ::= <char>
         | '\' <char>
         | '(' <regex> ')'
```

Informally, this translates to:

- A regular expression is a term;
  or a regular expression is a term, a '|' and another regular expression.
- A term is a possibly empty sequence of factors.
- A factor is a base followed by a possibly empty sequence of '*'.
- A base is a character, an escaped character, or a parenthesized regular expression.

A recursive descent parser for regex

A parser constructs the tree structure of a term.

Programmers often use tools like yacc to convert grammars to parsers.

When these tools are unavailable or inappropriate, one can parse manually using a technique like recursive descent.

Not all grammars are suitable for recursive descent, but many can be made suitable with refactoring, usually to eliminate left recursion in the grammar.

The core idea in recursive descent is to construct a procedure for each kind of term in the grammar.

Under the provided regular expression grammar, this means we need four procedures: `regex()`, `term()`, `factor()` and `base()`.

Each procedure parses a term of that type off the current input stream.

These procedures can call each other, and to interact with the input stream, they can call three primitives: `peek()`, `next()` and `eat()`:

- `peek()` returns the next item of input without consuming it;
- `next()` returns the next item of input and consumes it; and
- `eat(`*`item`*`)` consumes the next item of input, failing if not equal to *item*.

At a high level, the structure of the parser is:

```
/*
 A data type to represent a regular expression.
 */
abstract class RegEx {...}


/*
 A parser to be constructed each time
 a regular expression needs to be parsed.
 */
class RegExParser {

  public RegExParser(String input) {...}

  public RegEx parse () {...}

  /* Recursive descent parsing internals. */

  private char peek() {...}
  private void eat(char c) {...}
  private char next() {...}


  /* Regular expression term types. */

  private RegEx regex() {...}
  private RegEx term() {...}
  private RegEx factor() {...}
  private RegEx base() {...}
}
```

Recursive descent primitives

To set up the parsing object, we need to store its the input string internally. That's precisely what the constructor does:

```
  private String input ;

  public RegExParser(String input)  {
    this.input = input ;
  }
```

The recursive descent primitives can assume access to `input`:

```
  private char peek() {
    return input.charAt(0) ;
  }

  private void eat(char c) {
    if (peek() == c)
      this.input = this.input.substring(1) ;
    else
      throw new
       RuntimeException("Expected: " + c + "; got: " + peek()) ;
  }

  private char next() {
    char c = peek() ;
    eat(c) ;
    return c ;
  }

  private boolean more() {
    return input.length() > 0 ;
  }
```

The nonstandard primitve `more()` checks if there is more input available.

If efficiency were a concern, we would bump an index through an array rather than compute the substring at every step.

Regular expression parsing primitives

There are formal rules for constructing recursive descent parsers, but one can often intuit them by trying to fill in the blank procedures.

regex()

For `regex()` method, we know that we must parse at least one term, and whether we parse another depends only on what we find afterward:

```
  private RegEx regex() {
    RegEx term = term() ;

    if (more() && peek() == '|') {
      eat ('|') ;
      RegEx regex = regex() ;
      return new Choice(term,regex) ;
    } else {
     return term ;
    }
  }
```

To record the struture of the choice operation, we introduce a new subclass of `RegEx`:

```
class Choice extends RegEx
{
  private RegEx thisOne ;
  private RegEx thatOne ;

  public Choice (RegEx thisOne, RegEx thatOne) {
    this.thisOne = thisOne ;
    this.thatOne = thatOne ;
  }
}
```

term()

`term()` has to check that it has not reached the boundary of a term or the end of the input:

```
  private RegEx term() {
    RegEx factor = RegEx.blank ;

    while (more() && peek() != ')' && peek() != '|') {
      RegEx nextFactor = factor() ;
      factor = new Sequence(factor,nextFactor) ;
    }

    return factor ;
  }
```

To record the concatenation, we need another subclass of `RegEx`:

```
class Sequence extends RegEx
{
  private RegEx first ;
```

```
  private RegEx second ;

  public Sequence (RegEx first, RegEx second) {
    this.first = first ;
    this.second = second ;
  }
}
```

To record the empty regular expression, we need a different subclass:

```
class Blank extends RegEx {
}
```

Since there is only one kind of blank, `RegEx.blank` is a static constant.

factor()

To implement factor, we parse a base and then any number of Kleene stars:

```
  private RegEx factor() {

    RegEx base = base() ;

    while (more() && peek() == '*') {
      eat('*') ;
      base = new Repetition(base) ;
    }

    return base ;
  }
```

This clearly requres a subclass of `RegEx` to capture repetition:

```
class Repetition extends RegEx
{
  private RegEx internal ;

  public Repetition(RegEx internal) {
    this.internal = internal ;
  }
}
```

base()

The implementation of `base()` checks to see which of the three cases it has encountered:

```
  private RegEx base() {

    switch (peek()) {
      case '(':
        eat('(') ;
        RegEx r = regex() ;
        eat(')') ;
      return r ;

      case '\\':
       eat ('\\') ;
       char esc = next() ;
      return new Primitive(esc) ;

      default:
      return new Primitive(next()) ;
    }
  }
```

The last subclass of `RegEx` holds an individual character:

```
class Primitive extends RegEx
{
  private char c ;

  public Primitive(char c) {
    this.c = c ;
  }
}
```

Conclusion

A grammar guides the design of a recursive-descent parser.

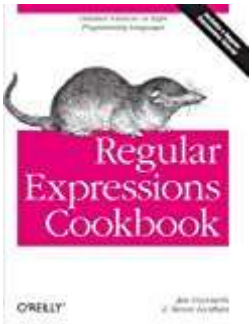
In the case of regular expressions, we are able to synthesize a tree representing the structure of a regular expression with straightforward, hand-written code.

From here, it's a fun exercise to implement `derive()`, `matches()` and `isNullable()` methods in the `RegEx` class (or subclasses as appropriate) using derivatives.

More resources on regex

A separate post covers regular expressions in the context of Unix text-manipulation tools.

For a solid "how-to" book on regular expressions themselves, I recommend Regular Expressions Cookbook:



Related pages

- Desugaring regular operations in context-free grammars
- Grammar: The language of languages (BNF, EBNF, ABNF)
- Standalone lexers with lex: synopsis, examples, and pitfalls
- Parsing with derivatives (Yacc is dead: An update)
- A non-blocking lexing toolkit for Scala from regex derivatives
- Lexical analysis and syntax-highlighting in JavaScript
- Matching regular expressions with derivatives
- Implementing regular expressions and NFAs in Java
- Parsing M-Expressions in Scala with combinators

[article index] [email me] [@mattmight] [+mattmight] [rss]

Latest: HOWTO: Get tenure
Next: Writing an interpreter, CESK-style
Prev: Hunting down my son's killer
Rand: Logical literacy