Parsing

Eric McCreath

Overview

In this lecture we will look at:

- structured text,
- generation,
- parsing,
- tokens,
- grammars, and
- writing a simple parser.

Structured Text

- Information is often stored in text files. Also we often provide information to a computer via text. Examples include:
  - html, xml
  - java, c, c++, haskell, perl, php, etc
  - mathematical expressions
  - web searches
- These linear textual representations are structured. We have agreed rules for writing and reading them. Such structure can be generally interpreted as a tree.

Generation/Parsing

- Generation involves creating the linear textual representation from the tree representation.
- This is simply a matter of traversing the tree and outputting the result as the traversal progresses.
- Parsing is the inverse of this operation. It involves taking the linear textual representation and generating the tree (note in some cases no explicit tree is generating, rather, a side effect is calculating as the tree is traversed).
- Parsing is generally more complex than generation.

Tokenization

The first step in parsing involves forming the text into a stream of basic tokens. This is known as tokenizing the text.

So suppose we are parsing the text:

```
inc(inc(0))
```

tokenizing would generate the sequence of tokens:

```
"inc", "(", "inc", "(",  0, ")", ")"
```

This simplifies the work of the next stage of the parsing process. As it will generally remove white space and parse basic elements, such as integers and doubles.

Tokenization

The tokenization process is a linear operation and the tokens are generally only generated as they are consumed by the parser, so it only requires a sequential read of the input text.

Tokenization is often implemented using a "finite state machine". As individual characters are read from the stream the tokenizing code is in one of a finite set of states. The states include states like "reading a label", "just read a <", or "reading a number". Different characters make the current state change and also return tokens.

Grammars

- Grammars provide a precise way of specifying the linear textual representation. A context free grammar is often used to define the syntax (Backus-Naur form is commonly used). A context free grammar is specified via a set production rules. This involves:
  - variables (surrounded with **<>**),
  - terminals (in quotes **" "**),
  - alternatives (**|**), and
  - production rules (which have the form **X ::= Y** where variable **X** may be replaced with **Y**).

For more info see:

https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

Also for a more expressive way of specifying a grammar EBNF (Extended Backus Form) is often used.

Grammars

```
<sentence> ::= "The " <animal> " sat on the mat."
<animal> ::= "cat" | "dog" | "mouse"
```

would specify the language:

```
{"The cat sat on the mat.", "The dog mat on the mat.",
 "The mouse sat on the mat."}
```

whereas

```
<num> ::= <digit> <num> | <digit>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

would specify the language:

```
{0,1,2,3,4,5,6,7,8,9,10,11,12,....}
```

Note, this grammar would include numbers like **003** , this could be fixed by modifying the grammar. That is if you didn't want to include numbers like **003** .

Parse Trees

The production rules are replacement rules so a variable on the left hand side of the production rule can be replaced by one of the alternatives on the right hand side.

so with the grammar below:

```
<sentence> ::= "The " <animal> " sat on the mat."
<animal> ::= "cat" | "dog" | "mouse"
```

we start with:

```
<sentence>
```

which is replaced with:

```
"The " <animal> " sat on the mat."
```

then animal is replace with on of the alternatives, say "mouse", which produces:

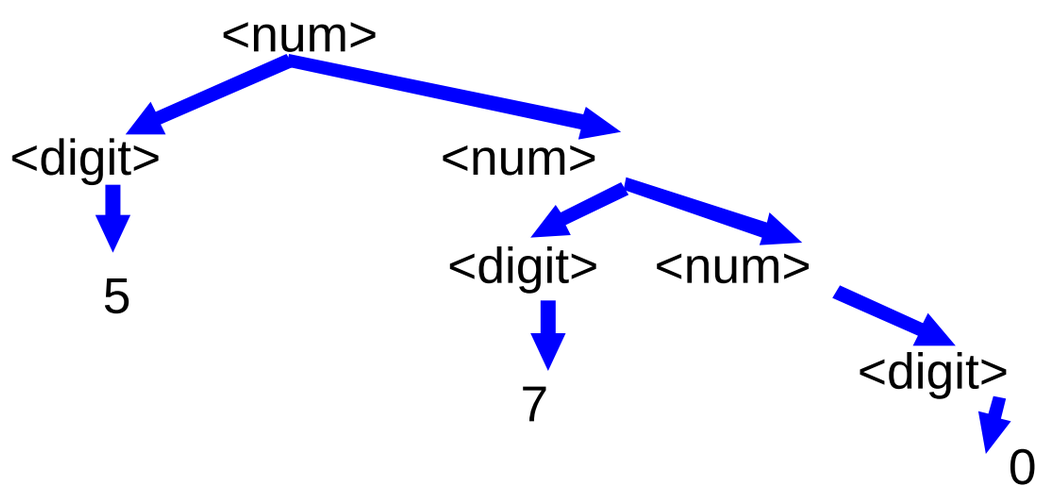
```
"The mouse sat on the mat."
```

Parse Trees

One can sequentially apply production rules until only terminal symbols remain. e.g.

```
<num>
<digit><num>
<digit><digit><num>
<digit><digit><digit>
5<digit><digit>
57<digit>
570
```

the variables and terminals can be thought of as nodes of a tree. This is know as a parse tree.



If there is a string that has different possible parse trees then the language is said to be ambiguous.

Implementing a parser

One can often simply implement a top down recursive descent parser for a grammar. This involves:

- creating a method for each production rule in the grammar,
- these methods are responsible for generating what is required for the variable on the left hand side of the production rule,
- the method consumes tokens in a left to right fashion:
  - if the production rule has terminal symbols then this terminal can be consumed directly from the tokenizer,
  - if the production rule has variables then it recursively calls their associated methods,
  - if the production rule has alternatives then work out which alternative to follow, and follow it.

Parsing Example

Say we have the grammar:

```
<exp> ::=  "inc" "(" <exp> ")" | "dec" "(" <exp> ")" | <num>
```

we could create the method:

```
Expression parseExpression(Tokenizer t) {
     if (t.current().equals("inc")) {
          t.next();
          t.parse("(");
          Expression subexp = parseExpression(t);
          t.parse(")");
          return new IncExpression(subexp);
     else if (t.current().equals("dec")) {
          t.next();
          t.parse("(");
          Expression subexp = parseExpression(t);
          t.parse(")");
          return new IncExpression(subexp);
     else if (t.current() instanceof Integer) {
          Integer v = (Integer) t.current();
          t.next();
          return new IntegerExpression(v);
     }
}
```

Recursive Descent Parser - Limitations

- The recursive descent parsing approach will not work with left recursive grammars. So a grammar like:

```
<binary> ::= <binary><digit>  | <digit>
<digit> ::= "0" | "1"
```

could not be parsed using the simple recursive descent approach. However we could transform the grammar into:

```
<binary> ::= <digit><binary> | <digit>
<digit> ::= "0" | "1"
```

which represents the same language, yet, is parsable using the predictive top-down approach.

Recursive Descent Parser - Limitations

- Also as grammars get more complex writing the code for the parser becomes tedious. So often people will use tools that automatically generate such code. These tools may also generate code for more complex bottom-up parsers, which are often more flexible in terms of the grammars they can deal with, although considerably more difficult to implement by hand.

See:
https://en.wikipedia.org/wiki/LR_parser
https://en.wikipedia.org/wiki/Recursive_descent_parser