An Introduction To Parse::RecDescent

Abram Hindle

Victoria Perl Mongers

abez@abez.ca

August 17, 2004

This Presentation

- What should you expect from this presentation?
  - Tutorial on writing a grammar for Parse::RecDescent
  - Recommended Action plans for a interpretter
  - Tips regarding grammars and what to avoid
  - Some introduction into into Parse::RecDescent special features.
  - How do deal with newlines.

This Presentation

- What should you NOT expect from this presentation?
  - A Course of Compilers and Interpretters
  - In depth discussion of grammars - why some things don't work
  - Theory of Grammars and parsing. Just remember that it's quite complex.

Parse::RecDescent

- What is Parse::RecDescent?
  - Generate Recursive-Descent Parsers [Con97]
  - Can Produces a parse tree
  - Parses a grammar recursively.
    - It can handle things like $(x + y * (x + y * (x + y * (x + y * (x)))))$
  - Can parse input text into a Concrete Syntax Tree and modify into a Abstract Syntax Tree (manually)
  - Similar to Yacc, Bison, Lex, SableCC

Terminology

- Language - a formal definition of the structure of text we want to parse
- Grammar - The definition of a language (not all languages have appropriate grammars compatible with Parse::RecDescent)
- Recursive - multiple levels multiple calls
- Parser - a program which reads in text in a language and produces some output (usually more program friendly)
- Semantic Analysis - Makes sure what was written in the language makes sense - allows us to have easier grammars e.g. should $(x + y).length()$ be allowed?
- Expression - a statement in a language which has a value
- Statement - a command in a language doesn’t necessarily return a value.

Languages

- Why parse languages?
  - Read formatted data.
  - Read human formatted data or provide a human interface to a complex problem.
  - Create a source code analyzers
  - Create a interpretter
  - Create a compiler

Grammar

- Grammars are built from rules
  - A rule is given a name and then a '—' delimited list of possible matches to the rule. Regular expressions are allowed.
  - It is recommended you seperate elements into easily parsable blocks
  -
```
quotedstring    :    /"(([^"]*)(\\["\\])?)*"/
identifier      :    /[a-z]\w+/
select          :    'select' identifier 'from' identifier 'where
                |    'select' identifier 'from' identifier 'where
```

Grammar

- Grammars are built from rules
  - Rules can be Right recursive
  - ```
    identifier  :  /[a-z]\w+/
    binops      :  '+' | '-' | '/' | '*'
    number      :  /(\d+|\d*\.\d+)/
    expression  :  identifier binops expression
                |  number binops expression
                |  identifier
                |  number
    ```
    ```

Grammar

- Grammars are built from rules
  - Rules can but not necessary left recursive (it is possible just you have to be very careful).
  - # won't work

```
identifier    :    /[a-z]\w+/
binops        :    '+' | '-' | '/' | '*'
number        :    /(\d+|\d*\.\d+)/
expression    :    expression binops identifier
              |    expression binops number
              |    identifier
              |    number
```

Grammar

- Lets take a look at a simple grammar
  - ```
    typeID          :       /[A-Z]\w+/
    attribID        :       /[a-z]\w+/
    quotedstring    :       /"(([^"]*)(\\")?)*"/

    startrule       :       definition
    definition      :       typeID '{'  statement(s)  '}'
    statement       :       attribID quotedstring <skip:'[ \t]*'>
         newline
    newline         :       "\n"
    ```
    ```

Grammar

- What can Grammar1 parse? Eg..
  - `statement`
    ```
    name  "White␣Mana"
    ```
  - `startrule`
    ```
    Energy {
            name  "White␣Mana"
            id  "white"
    }
    ```

Grammar

- Lets refine our grammar a bit..
  - ```
    typeID          :       /[A-Z]\w*/
    attribID        :       /[a-z]\w*/
    quotedstring    :       /"(([^"]*)(\\")?)*"/
    identifier      :       /[a-zA-Z]\w*/

    startrule       :       definition
    definition      :       typeID '{'  statement(s)  '}'
    statement       :       attribID quotedstring <skip:'[␣\t]*'>
         newline
                            | attribID codeblock
    newline         :       "\n"
    codeblock       :       '{' expression(s) '}'
    expression      :       methodcall
                    |       property
                    |       identifier
                    |       functioncall
    methodcall      :       identifier '.' identifier '(' expression(s
    ```
    ```

```
    ? /,/) ')'
                |       '(' expression ')' '.' identifier '('
                      expression(s? /,/) ')'
property        :       identifier '.' identifier
callable        :       methodcall
                |       property
functioncall    :       identifier '(' expression(s? /,/) ')'
```

Grammar

- What can Grammar2 parse?
  - The previous Grammar 1 examples as well as
  - `codeblock`

    ```
    { x.has(flying) }
    ```
  - `definition`

    ```
    Attribute {
            name  "flying"
            blockable { x.has(flying) }
    }
    ```
  - `definition`

    ```
    Attribute {
            name  "flying"
            blockable {
                    x.has(flying)
    ```

```
                (x.getChildren()).has(flying)
        }
}
```

Grammar

- What where some added features?
  - expression(s) - this means 1 or more expressions
  - expression(s? /,/) - 0 or more expressions delimited by commas
  - the skip directive changed the characters that we were skip over - whitespace is always skipped unless you are explicit with skip. We took out newlines so we could use it as a delimiter.

Grammar

- Lets refine our grammar a bit more. Lets add binary operators.

```
– number         :    /(\d+|\d*\.\d+)/
  typeID         :    /[A-Z]\w*/
  attribID       :    /[a-z]\w*/
  quotedstring   :    /"(([^"]*)(\\")?)*"/
  identifier     :    /[a-zA-Z]\w*/
                      {[@item]}
  lbinops        :    '->'| '==' | '>=' | /<=>?/ | '>' | '<' | '
       !='
  nbinops        :    '+' | '-' | '/' | '*'

  startrule      :    definition
  definition     :    typeID '{'  statement(s)  '}'
                      {[@item] }
  statement      :    attribID quotedstring <skip:'[␣\t]*'>
       newline
                      {[@item] }
                 |    attribID codeblock
```

```
                        {[@item] }
newline         :       "\n"
codeblock       :       '{' expline(s?) '}'
                        {[@item] }
expline         :       expression
expression      :       or_expr
sexpression     :       functioncall
                |       loop
                |       conditional
                |       property
                |       number
                |       identifier

or_expr         :       and_expr '||' or_expr {[@item] }
                |       and_expr
and_expr        :       lbinop_expr '&&' and_expr {[@item] }
                |       lbinop_expr
lbinop_expr     :       nbinop_expr lbinops lbinop_expr {[@item] }
                |       nbinop_expr
nbinop_expr     :       not_expr nbinops nbinop_expr {[@item] }
                |       not_expr
```

```
not_expr         :       '!' expression {[@item] }
                 |       methodcall
                 |       brack_expr
                 |       sexpression
brack_expr       :       '(' expression ')' { [@item[0,2]] }

loop             :       foreach
                 |       while

conditional      :       ifstatement

elseifstatement :        'else' 'if' '(' expression ')' codeblock
                         {[@item]}
elsestatement :          'else' codeblock
                         {[@item]}
ifstatement      :       'if' '(' expression ')' codeblock
     elseifstatement(s?) elsestatement(?)
                         {[@item]}
methodcall       :       identifier '.' identifier '(' expression(s
     ? /,/) ')'
                         {[@item] }
```

```
                |       brack_expr '.' identifier '(' expression(s
                      ? /,/) ')'
                        {[@item] }

property        :       identifier '.' identifier
                        {[@item] }
callable        :       methodcall
                |       property
functioncall    :       identifier '(' expression(s? /,/) ')'
                        {[@item] }
foreach         :       'foreach' '(' expression ')' '{' expression
     (0..) '}'
                        {[@item] }
                |       'foreach' '(' expression ')' 'st' '('
                      expression ')' '{' expression(0..) '}'
                        {[@item] }
                |       'foreach' identifier '(' expression ')' '{'
                       expression(0..) '}'
                        {[@item] }
                |       'foreach' identifier '(' expression ')' 'st
                      ' '(' expression ')' '{' expression(0..) '}'
```

```
                        {[@item] }
while        :          'while' '(' expression ')' '{' expression
     (0..) '}'
                        {[@item] }
```

Grammar

- What where some added features?
  - We told the parser what to keep and what not to keep.

    ```
    {[@item]} #this copies all the parse elements into an array ref.
    ```

  - We added precedence. See the or_expr coming before add_expr.
  - Notice the large chain.. That was created to deal with precedence and composing expressions of expressions.

Grammar

- This is what we expect our new grammar to parse.
  - ```
    definition
    Attribute {
            name  "flying"
            blockable { x.has(flying) }
    }
    ```
  - ```
    Attribute {
            name  "trample"
            event(attack.end) {
                    a = self.getAttacks
                    foreach (a) {
                            a.targetPlayer.mDamage(max(0,self.attack -
                                 a.blockedTotal))
                    }
            }
    }
    ```

- definition
```
Attribute {
        name  "flying"
        blockable {
                x.has()
                x.has(flying)
                y.has(flying)
                functioncall1(flying)
                functioncall2()
        }
}
```

- statement
```
blockable {
        x.has()
}
```

- statement
```
blockable {
        x.has()
```

```
        x.has(flying)
        y.has(flying)
        functioncall1(flying)
        functioncall2()
}
```

- codeblock

```
{
                x.has()
                x.has(flying)
                y.has(flying)
                functioncall1(flying)
                functioncall2()
}
```

- codeblock

```
{
        what + what
        what - what
        what - what + func(what)
```

```
        what - what + func(1)
}
```

- codeblock

```
{
        (chooseTargetPlayer()||chooseTargetCreature())
        (chooseTargetPlayer()||chooseTargetCreature()).methodCall
             (1)
}
```

Grammar Optimization

- But it’s so slow!
  - Simplify your grammar. Make the parser do less and less checking and be smarter how you parse your tree.
  - My method calls are extremely slow those could be greatly improved in speed by abstracting the method call operator ’.’ into a binary op.
  - REMEMBER your semantic check can be used to filter out the bad cases. The only problem is that by the semantic check usually you’ve lost the line numbers.

Parse::RecDescent

- How do we use it?
  - Here's an example

```
#!/usr/bin/perl
use Parse::RecDescent;
use Data::Dumper;
$::RD_TRACE = 1 if $ARGV[3];
$::RD_AUTOACTION = q { print "[",join("][",@item),"]",$/; [@item
     ] } if $ARGV[2];

open(FILE,$ARGV[0]) or die "$ARGV[0]␣not␣found";
my @grammar = <FILE>;
close(FILE);
open(FILE,$ARGV[1]) or die "$ARGV[0]␣not␣found";
my @input = <FILE>;
close(FILE);
my $startrule = shift @input;
chomp($startrule);
my $parser = new Parse::RecDescent("@grammar");
```

```
print Dumper( $parser->$startrule("@input") );
```

Parse::RecDescent

- How do we use it?
  - Parse your grammar as a scalar..
  - Input your text into it as a starting rule. In this case startrule is a rule in the grammar.
  - ```
    use Parse::RecDescent;
    my $parser = Parse::RecDescent->new($grammar);
    my $out = $parser->startrule($input);
    ```
    ```

Parse::RecDescent

- How do we debug it?
  - Turn on

```
$::RD_TRACE = 1;
```

Parse::RecDescent

- How do we get data out?
  - We want to prune the tree or get the tree?
  - A simple way to generate a tree is AUTOACTION

```
$::RD_AUTOACTION = q{ print "[",join("][",@item),"]",$/; [@item] }
```

Parse::RecDescent

- I have my tree.. what do I do now?
  - Semantic Analysis
  - Pruning (remove unnecessary nodes)
  - Store it
  - Bind to objects (recursively descend and generate an executable tree)
  - Binds to objects who output code (compiler)
  - Remember you’ll probably need a symbol table.

Parse::RecDescent

- Why Semantic Analysis?
  - Typechecking and symbol checking
    - $x = y + z$ - what if z is an object and x an integer?
  - Imagine a methodcall hack
    - $x.retInt().length()$ - if retInt returns an int primitive maybe length can't be called on it.
  - Maybe you want to limit syntax?

Parse::RecDescent

- Tree ⟵ Object
  - Recursive Function
  - Use composite pattern where you ask for values from child objects.
  - Assign objects in a tree using a composite pattern.
    - have a composite base class for these objects
    - have a recursive composite method to execute the children

Parse::RecDescent

- Misc Problems
  - Left associativity
  - Precedence
  - ambigous grammar
  - Left Recursion

Help!

- How to get help?
  - perldoc Parse::RecDescent
  - PerlMonks
  - Resources about writing compilers using Yacc, Bison, SableCC etc.
  - SableCC is quite similar to the Parse::RecDescent

Summary

- Make your grammar but be careful
- Either bind your grammar to objects or parse your AST.
- After parsing the tree you should attach it to objects and execute that or compile
- Interpretters are easy, compilers are much harder.

References

[Con97] Damian Conway. Parse::recdescent - generate recursive-descent parsers. 1997.