10. PEGs, Packrats and Parser Combinators

Oscar Nierstrasz

Thanks to Bryan Ford for his kind permission to reuse and adapt the slides of his POPL 2004 presentation on PEGs. http://www.brynosaurus.com/
Roadmap

> Domain Specific Languages
> Parsing Expression Grammars
> Packrat Parsers
> Parser Combinators
Sources

> Parsing Techniques — A Practical Guide
  — [Chapter 15.7 — Recognition Systems]

> “Parsing expression grammars: a recognition-based syntactic foundation”

> “Packrat parsing: simple, powerful, lazy, linear time”
  — Ford, ICFP 02, doi:10.1145/583852.581483

> The Packrat Parsing and Parsing Expression Grammars Page:
  — http://pdos.csail.mit.edu/~baford/packrat/

> Dynamic Language Embedding With Homogeneous Tool Support
Roadmap

> Domain Specific Languages
> Parsing Expression Grammars
> Packrat Parsers
> Parser Combinators
> A DSL is a specialized language targeted to a particular problem domain
  — Not a GPL
  — May be *internal* or *external* to a host GPL
  — Examples: SQL, HTML, Makefiles
External DSL’s (Examples)

```vhdl
-- this is the entity
entity ANDGATE is
  port (  
    A : in std_logic;  
    B : in std_logic;  
    O : out std_logic);
end entity ANDGATE;

-- this is the architecture
architecture RTL of ANDGATE is
begin
  O <= A and B;
end architecture RTL;
```

```html
 pencolor white
 fd 100
 rt 120
 fd 100
 rt 120
 fd 100
 rt 60
 pencolor blue
 fd 100
 rt 120
 fd 100
 rt 120
 fd 100
 rt 60
```
A “Fluent Interface” is a DSL that hijacks the host syntax

Function sequencing

```
computer();
    processor();
        cores(2);
            i386();
    disk();
        size(150);
    disk();
        size(75);
        speed(7200);
    sata();
end();
```

Function nesting

```
computer(
    processor(
        cores(2),
        Processor.Type.i386),
    disk(
        size(150)),
    disk(
        size(75),
        speed(7200),
        Disk.Interface.SATA));
```

Function chaining

```
computer()
    .processor()
        .cores(2)
            .i386()
                .end()
    .disk()
        .size(150)
            .end()
    .disk()
        .size(75)
            .speed(7200)
            .sata()
                .end()
        .end();
```
Other approaches:
— Higher-order functions
— Operator overloading
— Macros
— Meta-annotations
— ...
An *embedded language* may adapt the syntax or semantics of the host language.

We will explore some techniques used to specify external and embedded DSLs.
Roadmap

- Domain Specific Languages
- Parsing Expression Grammars
- Packrat Parsers
- Parser Combinators
“Why do we cling to a generative mechanism for the description of our languages, from which we then laboriously derive recognizers, when almost all we ever do is recognizing text? Why don’t we specify our languages directly by a recognizer?”

Some people answer these two questions by “We shouldn’t” and “We should”, respectively.

— Grune & Jacobs, 2008
Designing a Language Syntax

Textbook Method

1. Formalize syntax via context-free grammar
2. Write a parser generator (.*CC) specification
3. Hack on grammar until “nearLALR(1)"
4. Use generated parser

What exactly does a CFG describe?

**Short answer:** a rule system to *generate* language strings

**Example CFG**

\[
S \rightarrow \textbf{aa}S \\
S \rightarrow \varepsilon
\]

output strings

start symbol
What exactly do we want to describe?

**Proposed answer:** a rule system to *recognize* language strings

**Parsing Expression Grammars** (PEGs) model recursive descent parsing best practice

Example PEG

\[ S \leftarrow aaS / \varepsilon \]

Key benefits of PEGs

> Simplicity, formalism of CFGs
> Closer match to syntax practices
  — More expressive than deterministic CFGs (LL/LR)
  — Natural expressiveness:
    - prioritized choice
    - syntactic predicates
  — Unlimited lookahead, backtracking
> Linear time parsing for any PEG (!)
Key assumptions

 Parsing functions must

1. be stateless - depend only on input string
2. make decisions locally - return one result or fail
Parsing Expression Grammars

A PEG $P = (\Sigma, N, R, e_S)$

- $\Sigma$: a finite set of terminals (character set)
- $N$: finite set of non-terminals
- $R$: finite set of rules of the form “$A \leftarrow e$”, where $A \in N$, and $e$ is a parsing expression
- $e_S$: the start expression (a parsing expression)
## Parsing Expressions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>the empty string</td>
</tr>
<tr>
<td>$a$</td>
<td>terminal ($a \in \Sigma$)</td>
</tr>
<tr>
<td>$A$</td>
<td>non-terminal ($A \in N$)</td>
</tr>
<tr>
<td>$e_1/e_2$</td>
<td>sequence</td>
</tr>
<tr>
<td>$e_1/e_2$</td>
<td>prioritized choice</td>
</tr>
<tr>
<td>$e^?, e^*, e^+$</td>
<td>optional, zero-or-more, one-or-more</td>
</tr>
<tr>
<td>$&amp;e, !e$</td>
<td>syntactic predicates</td>
</tr>
</tbody>
</table>

How PEGs express languages

> Given an input string $s$, a parsing expression $e$ either:
  
  — Matches and consumes a prefix $s'$ of $s$, or
  
  — Fails on $s$
Prioritized choice with backtracking

\[ S \leftarrow A / B \]

*means:* first try to parse an A. If A fails, then backtrack and try to parse a B.

\[ S \leftarrow \text{if } C \text{ then } S \text{ else } S \]

\[ / \text{if } C \text{ then } S \]

S matches “if C then S foo”
S matches “if C then S₁ else S₂”
S fails on “if C else S”
Greedy option and repetition

A ← e?  
is equivalent to  
A ← e / ε

A ← e*  
is equivalent to  
A ← e A / ε

A ← e+  
is equivalent to  
A ← e e*

I ← L+
L ← a / b / c / …

I matches "foobar"
I fails on "123"
Syntactic Predicates

& e succeeds whenever e does, _but consumes no input_

! e succeeds whenever e fails, _but consumes no input_

A ← foo & (bar)
B ← foo !(bar)

A matches “foobar”
A _fails_ on “foobie”
B matches “foobie”
B _fails_ on “foobar”
Example: nested comments

<table>
<thead>
<tr>
<th>Comment</th>
<th>←</th>
<th>Begin Internal* End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>←</td>
<td>!End ( Comment / Terminal )</td>
</tr>
<tr>
<td>Begin</td>
<td>←</td>
<td>/**</td>
</tr>
<tr>
<td>End</td>
<td>←</td>
<td>*/</td>
</tr>
<tr>
<td>Terminal</td>
<td>←</td>
<td>[any character]</td>
</tr>
</tbody>
</table>

C matches “/**ab*/cd”
C matches “/**a/**b*/c*/”
C fails on “/**a/**b*/”
Formal properties of PEGs

> Expresses all deterministic languages — LR(k)
> Closed under union, intersection, complement
> Expresses some non-context free languages
  — e.g., $a^n b^n c^n$
> Undecidable whether $L(G) = \emptyset$

Roadmap

- Domain Specific Languages
- Parsing Expression Grammars
- Packrat Parsers
- Parser Combinators
Top-down parsing techniques

**Predictive parsers**
- use lookahead to decide which rule to trigger
- fast, linear time

**Backtracking parsers**
- try alternatives in order; backtrack on failure
- simpler, more expressive (possibly exponential time!)
Example

Add ← Mul ± Add / Mul
Mul ← Prim * Mul / Prim
Prim ← ( Add ) / Dec
Dec ← 0 / 1 / … / 9

NB: This is a scannerless parser — the terminals are all single characters.
Parsing “6*(3+4)”

Add  ←  Mul ± Add / Mul
Mul  ←  Prim * Mul / Prim
Prim  ←  ( Add ) / Dec
Dec  ←  0 / 1 / … / 9
Memoized parsing: Packrat Parsers

> Formally developed by Birman in 1970s

By memoizing parsing results, we avoid having to recalculate partially successful parses.

```java
public class SimplePackrat extends SimpleParser {
    Hashtable<Integer, Result>[] hash;
    final int ADD = 0;
    final int MUL = 1;
    final int PRIM = 2;
    final int HASHES = 3;

    SimplePackrat (String input) {
        super(input);
        hash = new Hashtable[HASHES];
        for (int i=0; i<hash.length; i++) {
            hash[i] = new Hashtable<Integer, Result>();
        }
    }

    protected Result add(int pos) throws Fail {
        if (!hash[ADD].containsKey(pos)) {
            hash[ADD].put(pos, super.add(pos));
        }
        return hash[ADD].get(pos);
    }

    ...
}
```

Formally developed by Birman in 1970s
Memoized parsing “6*(3+4)”

Add <- Mul + Add
Mul <- Prim * Mul
Prim <- ( Add )
Char ()
Prim <- Dec [BACKTRACK]
Dec <- Num
Char 0
Char 1
Char 2
Char 3
Char 4
Char 5
Char 6
Char *
Mul <- Prim * Mul
Prim <- ( Add )
Char ()
Add <- Mul + Add
Mul <- Prim * Mul
Prim <- ( Add )
Char ()
Prim <- Dec [BACKTRACK]
Dec <- Num
Char 0
Char 1
Char 2
Char 3
Char 4
Char *
Mul <- Prim [BACKTRACK]
PRIM -- retrieving hashed result
Char +
Add <- Mul [BACKTRACK]
MUL -- retrieving hashed result
Char )
Char *
Mul <- Prim [BACKTRACK]
PRIM -- retrieving hashed result
Char +
Add <- Mul [BACKTRACK]
MUL -- retrieving hashed result
Eof
56 steps
6*(3+4) -> 42
What is Packrat Parsing good for?

> Linear cost
  — bounded by size(input) \times \#(parser rules)

> Recognizes strictly larger class of languages than deterministic parsing algorithms (LL(k), LR(k))

> Good for scannerless parsing
  — fine-grained tokens, unlimited lookahead

Scannerless Parsing

> Traditional linear-time parsers have fixed lookahead
  — With unlimited lookahead, don’t need separate lexical analysis!

> Scannerless parsing enables unified grammar for entire language
  — Can express grammars for mixed languages with different lexemes!
What is Packrat Parsing *not* good for?

- General CFG parsing (ambiguous grammars)
  — produces at most one result

- Parsing highly “stateful” syntax (C, C++)
  — memoization depends on statelessness

- Parsing in minimal space
  — LL/LR parsers grow with stack depth, not input size

Roadmap

- Domain Specific Languages
- Parsing Expression Grammars
- Packrat Parsers
- Parser Combinators
Parser Combinators

> Parser combinators in **functional languages** are higher order functions used to build parsers
  — e.g., Parsec, Haskell

> In an **OO language**, a combinator is a (functional) object
  — To build a parser, you simply compose the combinators
  — Combinators can be reused, or specialized with new semantic actions
    — *compiler, pretty printer, syntax highlighter* …
  — e.g., PetitParser, Smalltalk
PEG expressions are implemented by subclasses of PPParser. PEG operators are messages sent to parsers.

http://source.lukas-renggli.ch/petit.html
PetitParser example

| goal add mul prim dec |

dec := $0 - $9.
add := ( mul, $+ asParser, add ) / mul.
mul := ( prim, $* asParser, mul) / prim.
prim := ( $( asParser, add, $) asParser) / dec.
goal := add end.
goal parse: '6*(3+4)' asParserStream
  ⇒ #$($6 $* #$($(#($3 $+ $4) $)))}
Semantic actions in PetitParser

```
| goal add mul prim dec |

dec := ($0 - $9)
   ==> [ :token / token asNumber ]
add := ((mul, $+ asParser, add)
   ==> [ :nodes / nodes first + nodes third ])
   / mul.
mul := ((prim, $* asParser, mul)
   ==> [ :nodes / nodes first + nodes third ])
   / prim.
prim := ($($ asParser, add, $) asParser)
   ==> [ :nodes / nodes second ]
   / dec.
goal := add end.
```

Add ← Mul ± Add / Mul
Mul ← Prim * Mul / Prim
Prim ← (Add) / Dec
Dec ← 0 / 1 / … / 9

goal parse: '6*(3+4)' asParserStream ➜ 42
> Some OO parser combinator libraries:
  — Java: JParsec
  — C#: NParsec
  — Ruby: Ruby Parsec
  — Python: Pysec
  — *and many more* …
public class Calculator {

    ...
    static Parser<Double> calculator(Parser<Double> atom) {
        Parser.Reference<Double> ref = Parser.newReference();
        Parser<Double> unit = ref.lazy().between(term("("), term(")")).or(atom);
        Parser<Double> parser = new OperatorTable<Double>()
            .infixl(op("+", BinaryOperator.PLUS), 10)
            .infixl(op("-", BinaryOperator.MINUS), 10)
            .infixl(op("*", BinaryOperator.MUL).or(WHITESPACE_MUL), 20)
            .infixl(op("/", BinaryOperator.DIV), 20)
            .prefix(op("-", UnaryOperator.NEG), 30).build(unit);
        ref.set(parser);
        return parser;
    }

    public static final Parser<Double> CALCULATOR = calculator(NUMBER).from(
        TOKENIZER, IGNORED);

}
What you should know!

- Is a CFG a language recognizer or a language generator? What are the practical implications of this?
- How are PEGs defined?
- How do PEGs differ from CFGs?
- What problem do PEGs solve?
- How does memoization aid backtracking parsers?
- What are scannerless parsers? What are they good for?
- How can parser combinators be implemented as objects?
Can you answer these questions?

- Why is it critical for PEGs that parsing functions be stateless?
- Why do PEG parsers have unlimited lookahead?
- Why are PEGs and packrat parsers well suited to functional programming languages?
- What kinds of languages are scannerless parsers good for? When are they inappropriate?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

**Share** — copy and redistribute the material in any medium or format

**Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

**Under the following terms:**

**Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

**ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

[http://creativecommons.org/licenses/by-sa/4.0/](http://creativecommons.org/licenses/by-sa/4.0/)