

10. PEGs, Packrats and Parser Combinators

Prof. O. 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**
 - Grune & Jacobs, Springer, 2008
 - *[Chapter 15.7 — Recognition Systems]*
- > **“Parsing expression grammars: a recognition-based syntactic foundation”**
 - Ford, POPL 2004, doi:10.1145/964001.964011
- > **“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**
 - Renggli, PhD thesis, 2010, <http://scg.unibe.ch/bib/Reng10d>

Roadmap

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



Domain Specific Languages

- > 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

Internal DSLs

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

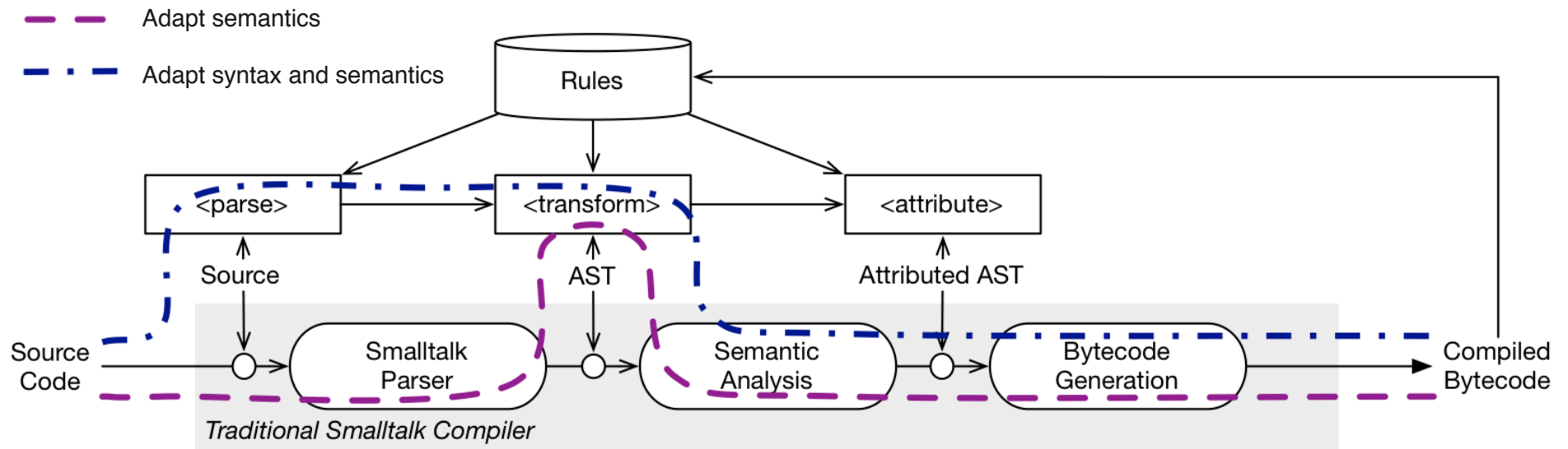
```
computer()
  .processor()
    .cores(2)
    .i386()
    .end()
  .disk()
    .size(150)
    .end()
  .disk()
    .size(75)
    .speed(7200)
    .sata()
    .end()
  .end();
```

Fluent Interfaces

- > *Other approaches:*
 - Higher-order functions
 - Operator overloading
 - Macros
 - Meta-annotations
 - ...

Embedded languages

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



Recognition systems

“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 YACC parser specification
3. Hack on grammar until “near-LALR(1)”
4. Use generated parser

Pragmatic Method

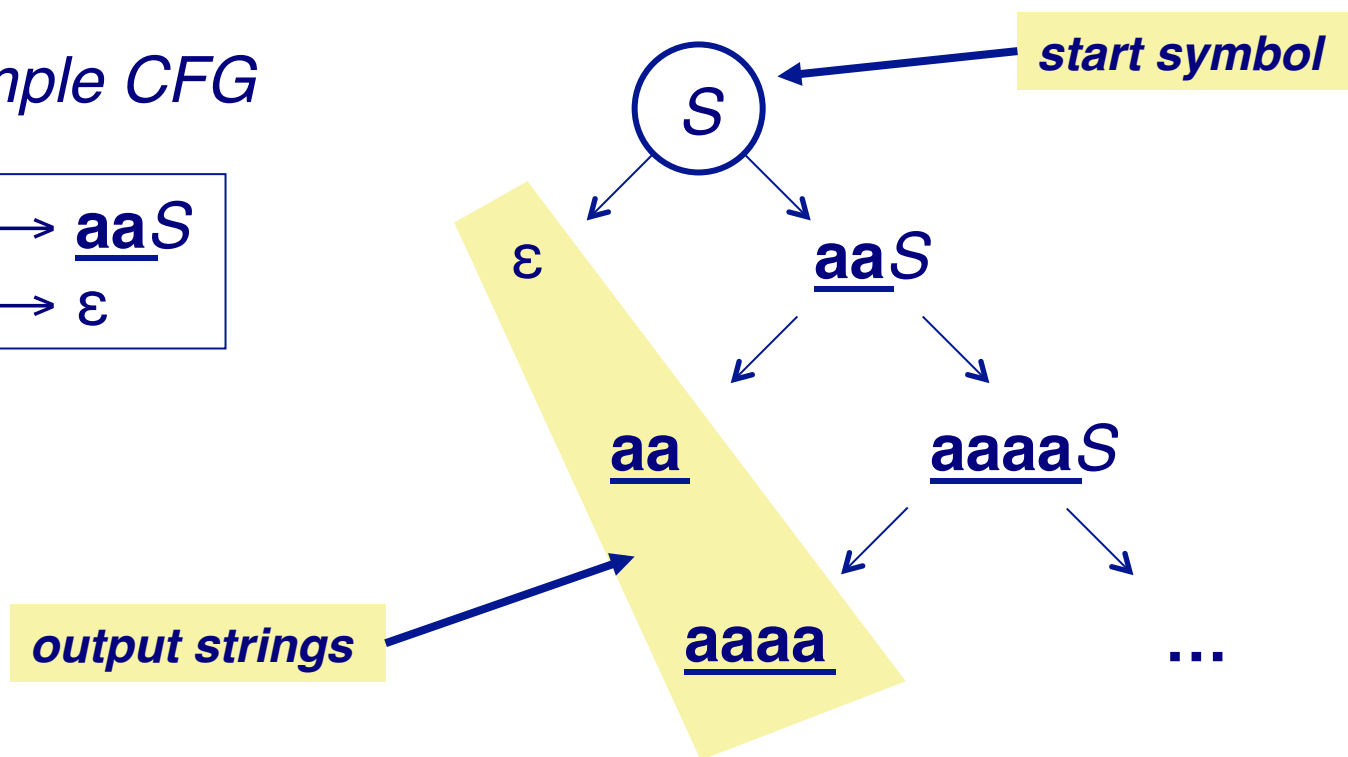
1. Specify syntax informally
2. Write a recursive descent parser

What exactly does a CFG describe?

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

Example CFG

$S \rightarrow \underline{aa}S$
 $S \rightarrow \epsilon$



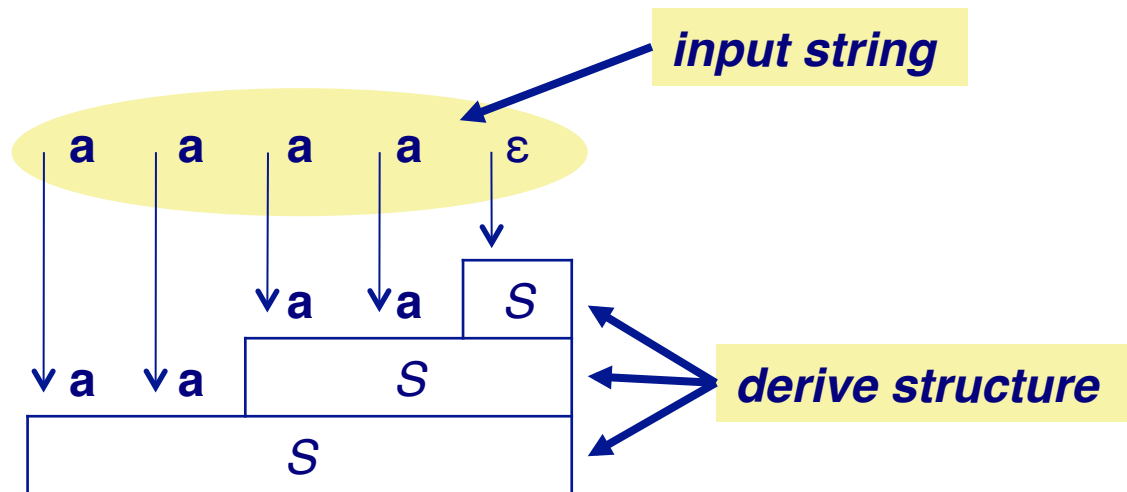
What exactly do we *want* to describe?

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

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

Example PEG

$S \leftarrow \underline{aa}S / \epsilon$



Key benefits of PEGs

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

Key assumptions

Parsing functions

- > must be stateless
 - depend only on input string
- > make decisions locally
 - return at most one result (success/failure)

Parsing Expression Grammars

- > A PEG $P = (\Sigma, N, R, e_S)$
 - Σ : 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

ε	the empty string
<u>a</u>	terminal (<u>a</u> $\in \Sigma$)
A	non-terminal (A $\in N$)
$e_1 e_2$	sequence
e_1 / e_2	prioritized choice
$e^?, e^*, e^+$	optional, zero-or-more, one-or-more
$\&e, !e$	syntactic predicates

How PEGs express languages

- > Given an input string s , a parsing expressing e either:
 - **Matches** and consumes a prefix s' of s , or
 - **Fails** on s

$S \leftarrow \underline{\text{bad}}$

S matches “badder”
 S matches “baddest”
 S *fails* on “**abad**”
 S *fails* on “**babe**”

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.

$$S \leftarrow \underline{\text{if } C \text{ then } S} \underline{\text{else } S} \\ / \underline{\text{if } C \text{ then } S}$$

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

$A \leftarrow e^?$	<i>is equivalent to</i>	$A \leftarrow e / \varepsilon$
$A \leftarrow e^*$	<i>is equivalent to</i>	$A \leftarrow e A / \varepsilon$
$A \leftarrow e^+$	<i>is equivalent to</i>	$A \leftarrow e e^*$

$I \leftarrow L^+$ $L \leftarrow \underline{a} / \underline{b} / \underline{c} / \dots$
--

I matches **foobar**
 I matches **foo(bar)**
 I *fails* on **123**

Syntactic Predicates

&e succeeds whenever e does, *but consumes no input*
!e succeeds whenever e fails

A ← foo &(bar)
B ← foo !(bar)

A matches “foobar”
A *fails* on “foobie”
B matches “foobie”
B *fails* on “foobar”

Example: nested comments

C	←	B I* E
I	←	!E (C / T)
B	←	(*
E	←	*
T	←	[<i>any terminal</i>]

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 $L(G) = \emptyset$

What can't PEGs express directly?

- > Ambiguous languages
 - That's what CFGs are for!
- > Globally disambiguated languages?
 - $\{\underline{a}, \underline{b}\}^n \underline{a} \{\underline{a}, \underline{b}\}^n$
- > State- or semantic-dependent syntax
 - C, C++ typedef symbol tables
 - Python, Haskell, ML layout

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 <u>+</u> Add / Mul
Mul	←	Prim <u>*</u> Mul / Prim
Prim	←	(Add) / Dec
Dec	←	<u>0</u> / <u>1</u> / ... / <u>9</u>

NB: This is a scannerless parser — the terminals are all single characters.

```
public class SimpleParser {
    final String input;
    SimpleParser(String input) {
        this.input = input;
    }
    class Result {
        int num; // result calculated so far
        int pos; // input position parsed so far
        Result(int num, int pos) {
            this.num = num;
            this.pos = pos;
        }
    }
    class Fail extends Exception {
        Fail() { super(); }
        Fail(String s) { super(s); }
    }
    ...
    protected Result add(int pos) throws Fail {
        try {
            Result lhs = this.mul(pos);
            Result op = this.eatChar('+', lhs.pos);
            Result rhs = this.add(op.pos);
            return new Result(lhs.num+rhs.num, rhs.pos);
        } catch(Fail ex) { }
        return this.mul(pos);
    }
    ...
}
```

Parsing “2*(3+4)”

```

Add <- Mul + Add
Mul <- Prim * Mul
Prim <- ( Add )
Char (
Prim <- Dec [BACKTRACK]
Dec <- Num
Char 0
Char 1
Char 2
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 *
Mul <- Prim [BACKTRACK]
Prim <- ( Add )
Char (
Prim <- Dec [BACKTRACK]
Dec <- Num

```

```

Char 0
Char 1
Char 2
Char 3
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 <- ( Add )
Char (
Prim <- Dec [BACKTRACK]
Dec <- Num
Char 0
Char 1
Char 2
Char 3
Char 4
Char +
Add <- Mul [BACKTRACK]

```

```

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 <- ( Add )
Char (
Prim <- Dec [BACKTRACK]
Dec <- Num
Char 0
Char 1
Char 2
Char 3
Char 4
Char )
Char *
Mul <- Prim [BACKTRACK]
...
Eof
304 steps

```

Memoization

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

```
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);
    }
    ...
}
```

Memoized parsing “2*(3+4)”

```

Add <- Mul + Add
Mul <- Prim * Mul
Prim <- ( Add )
Char (
Prim <- Dec [BACKTRACK]
Dec <- Num
Char 0
Char 1
Char 2
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 *
Mul <- Prim [BACKTRACK]
PRIM -- retrieving hashed result

```

```

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
52 steps
2*(3+4) -> 14

```

What is Packrat Parsing good for?

- > Formally developed by Birman in 1970s
 - but apparently never implemented
- > Linear cost
 - bounded by $\text{size}(\text{input}) \times \#(\text{parser rules})$
- > Recognizes strictly larger class of languages than deterministic parsing algorithms (LL(k), LR(k))
 - incomparable to class of context-free languages
- > Good for scannerless parsing
 - fine-grained tokens, unlimited lookahead

<http://www.brynosaurus.com/pub/lang/packrat-icfp02-slides.pdf>

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

- > A *combinator* is a (closed) higher-order function
 - used in mathematical logic to eliminate the need for variables
 - used in functional programming languages as a model of computation

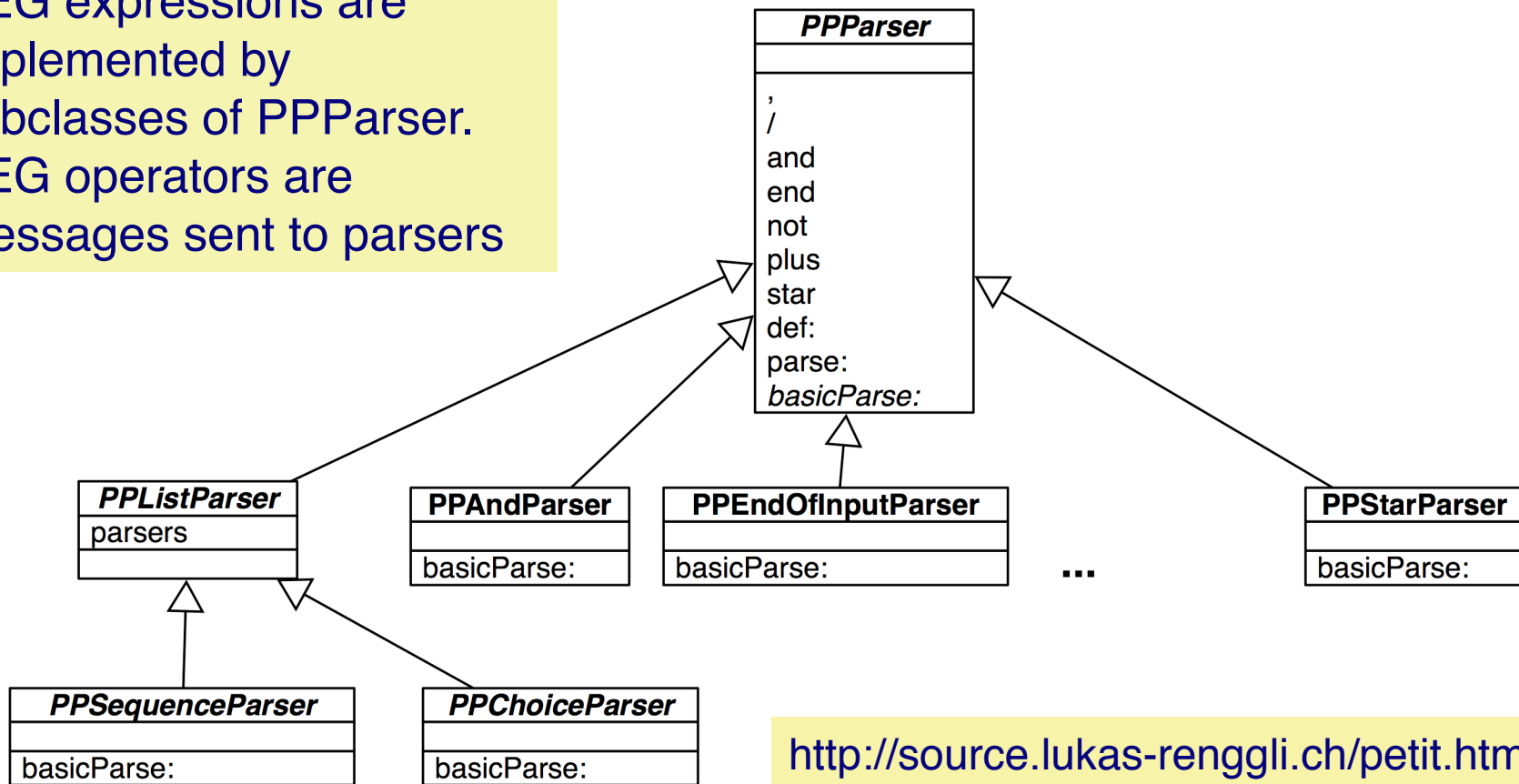
- > Parser combinators in functional languages are higher order functions used to build parsers
 - Parsec <http://www.haskell.org/haskellwiki/Parsec>

Parser Combinators in OO languages

- > 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 ...

PetitParser — a PEG parser combinator library for Smalltalk

PEG expressions are implemented by subclasses of `PPParser`. PEG operators are messages sent to parsers



PetitParser example

```
| goal add mul prim dec |
add := PPParser new.
mul := PPParser new.
prim := PPParser new.
```

```
dec := $0 - $9.
```

```
add def: ( mul, $+ asParser, add )
        / mul.
```

```
mul def: ( prim, $* asParser, mul )
         / prim.
```

```
prim def: ( $( asParser, add, $) asParser )
          / dec.
```

```
goal := add end.
```

```
goal parse: '2*(3+4)' asParserStream
           → #($2 $* #($ ( #($3 $+ $4) $)))
```

Add	←	Mul <u>+</u> Add / Mul
Mul	←	Prim <u>*</u> Mul / Prim
Prim	←	(Add) / Dec
Dec	←	<u>0</u> / <u>1</u> / ... / <u>9</u>

Semantic actions in PetitParser

```

| goal add mul prim dec |
add := PPParser new.
mul := PPParser new.
prim := PPParser new.
dec := ($0 - $9)
      ==> [ :token | token asciiValue - $0 asciiValue ].
add def: ((mul , $+ asParser , add)
          ==> [ :nodes | (nodes at: 1) + (nodes at: 3) ])
        / mul.
mul def: ((prim , $* asParser , mul)
          ==> [ :nodes | (nodes at: 1) * (nodes at: 3) ])
        / prim.
prim def: (($ ( asParser , add , $ ) asParser)
          ==> [ :nodes | nodes at: 2 ])
        / dec.
goal := add end.

goal parse: '2*(3+4)' asParserStream → 14

```

Parser Combinator libraries

- > Some OO parser combinator libraries:
 - Java: JParsec
 - C#: NParsec
 - Ruby: Ruby Parsec
 - Python: Pysec
 - *and many more ...*









Jparsec — composing a parser from parts

```
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>()
            .infix1(op("+", BinaryOperator.PLUS), 10)
            .infix1(op("-", BinaryOperator.MINUS), 10)
            .infix1(op("*", BinaryOperator.MUL).or(WHITESPACE_MUL), 20)
            .infix1(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);
}
```

<http://jparsec.codehaus.org/jparsec2+Tutorial>

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?*
-  *What are the formal limitations of PEGs?*
-  *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 do parser generators traditionally generate bottom-up rather than top-down parsers?*
-  *Why is it critical for PEGs that parsing functions be stateless?*
-  *How can you recognize the end-of-input as a PEG expression?*
-  *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?*
-  *How do parser combinators enable scripting?*

License

<http://creativecommons.org/licenses/by-sa/3.0/>



Attribution-ShareAlike 3.0 Unported

You are free:

- to Share** — to copy, distribute and transmit the work
- to Remix** — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.