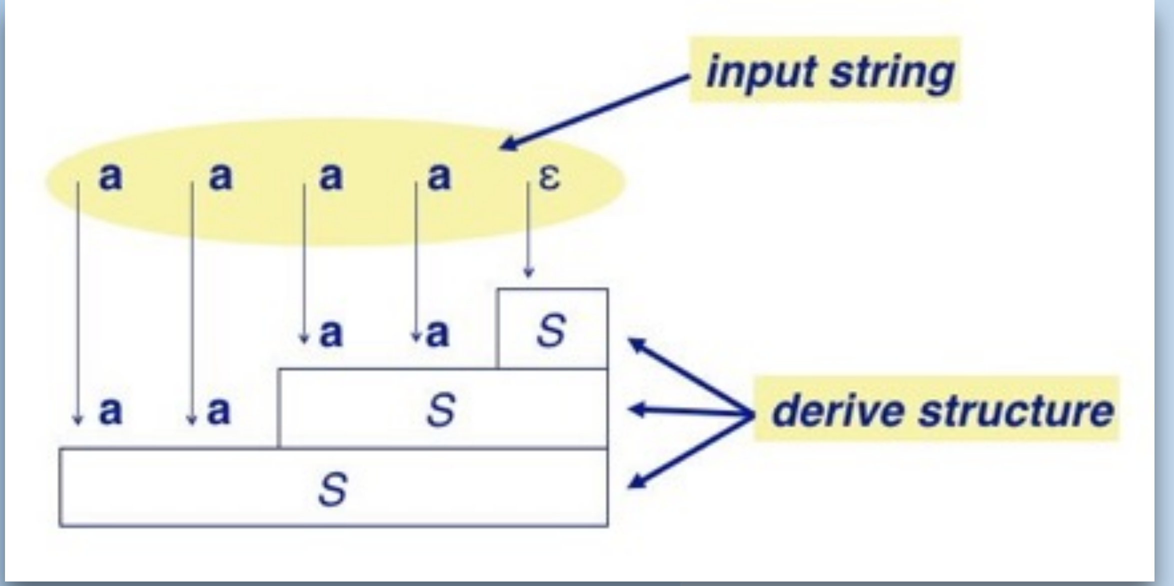
10. PEGs, Packrats and Parser Combinators

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/



UNIVERSITÄT BERN

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

A domain-specific language (DSL) is a language dedicated to a particular application domain. This domain may be technical (e.g., SQL) or it might be closer to the actual business domain.

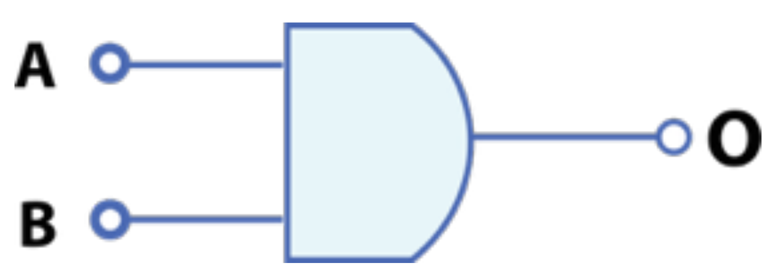
A DSL may be external, in which case it has its own syntax and implementation independent of the host language, or it may be internal, in which case it is embedded, and may even hijack the host-language syntax to emulate a language within a language.

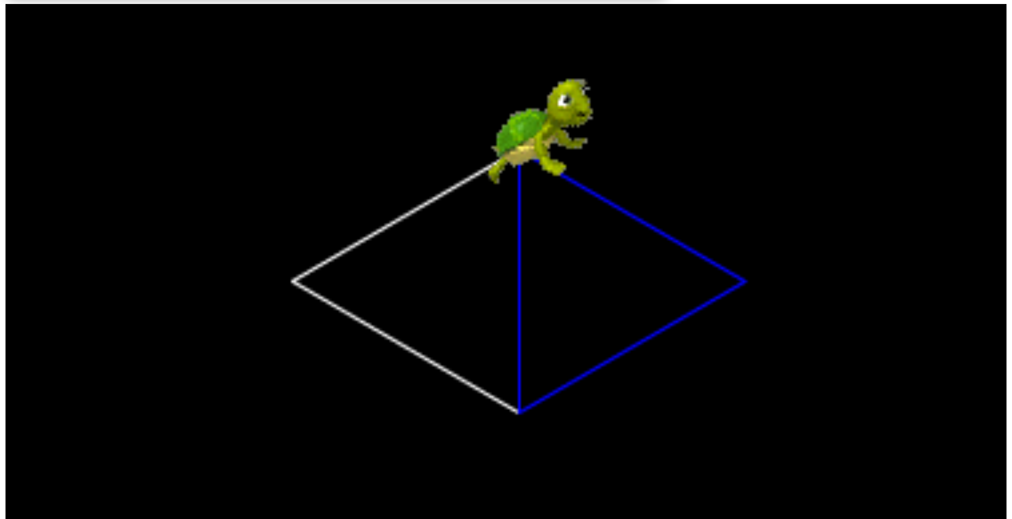
External DSLs (Examples)

```
-- 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;
```

```
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
```





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();
```

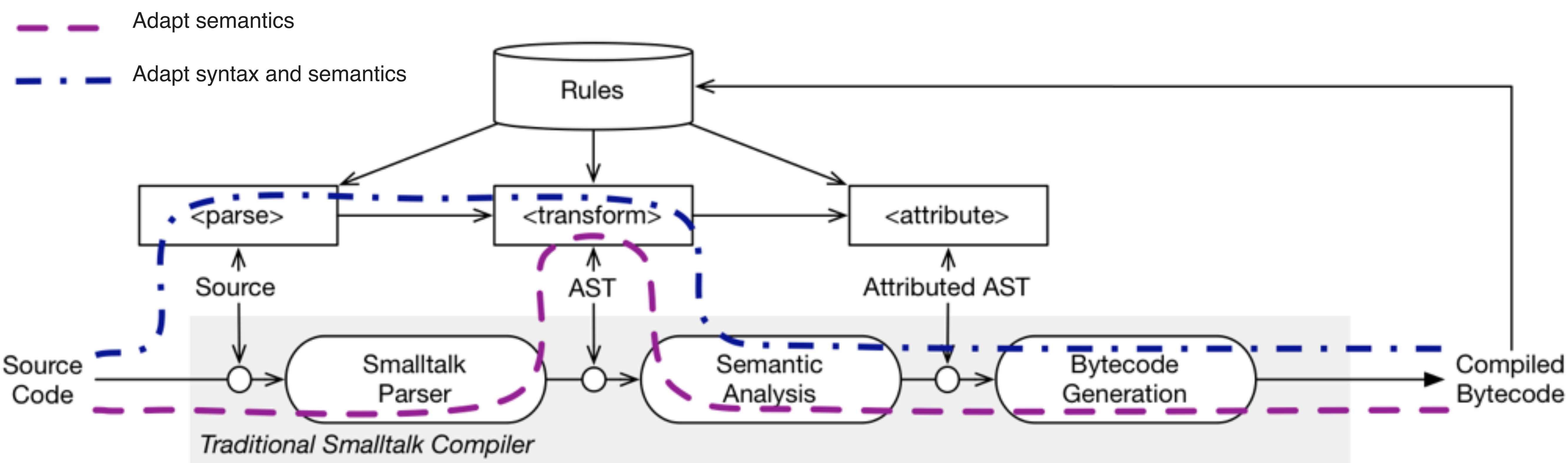
Fluent Interfaces

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

```
sizer.FromImage(i)
 .ReduceByPercent(x)
 .Pixalize()
 .ReduceByPercent(x)
 .OutputImageFormat(ImageFormat.Jpeg)
 .ToLocation(o)
 .Save();
```

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

Recall that Chomsky-style grammars define a language by the set of strings that they generate. Parsing then must go backwards to reverse engineer a parse for a given sentence in the language.

Designing a Language Syntax

Textbook Method

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

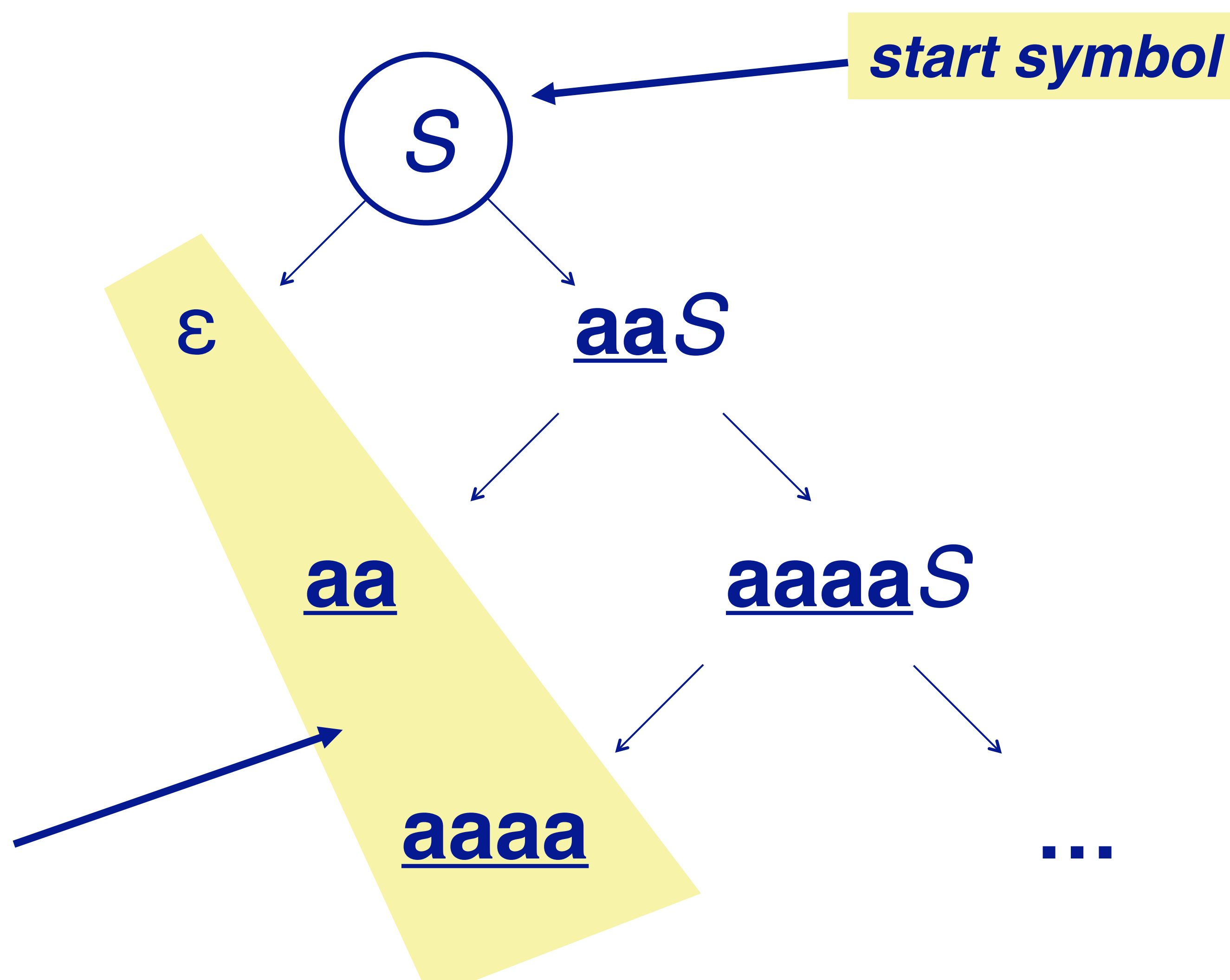
What exactly does a CFG describe?

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

Example CFG

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

$S \rightarrow \varepsilon$



output strings

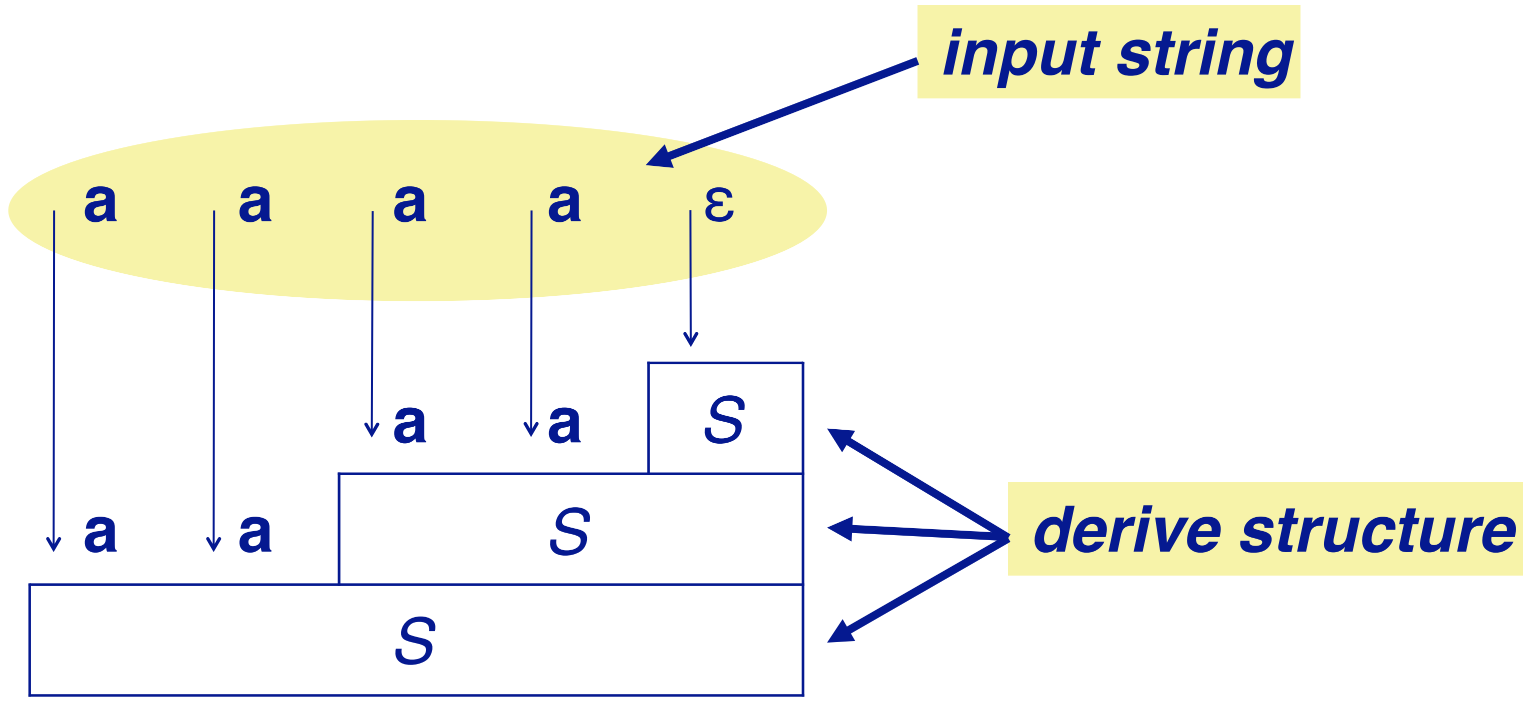
What exactly do we *want* to describe?

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

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

Example PEG

$S \leftarrow \underline{\mathbf{aa}}S\ /\ \varepsilon$



Unlike the CFG in the previous slide that generates sentences in a language, this PEG specifies rules to recognize sentences in a top-down fashion.

The "/" symbol represents an ordered choice. First we recognize "aa". This succeeds, so then we try to recognize S. Again we recognize "aa" and again recurse in S. This time "aa" fails, so we try to recognize ε. This succeeds, so we are done.

(In general we may fail and have to backtrack.)

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 (!)

As we shall see, linear parse time can be achieved with the help of memoization using a “packrat parser”.

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 = (Σ, 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 ← e", where A ∈ N, and e is a *parsing expression*
  - e_S : the *start expression* (a parsing expression)

Parsing Expressions

ε	the empty string
a	terminal (**a** $\in \Sigma$)
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

This looks pretty similar to a CFG with some important differences.

Choice is prioritized: e_1 / e_2 means first try e_1, then try e_2.

The syntactic predicates do not consume any input. &e succeeds if e would succeed, and !e succeeds if e would fail.

NB: "." is considered to match anything, so "!." matches the end of input.

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**

S ← **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.

$S \leftarrow$ **<u>if</u>** C **<u>then</u>** S **<u>else</u>** S
/ **<u>if</u>** C **<u>then</u>** S

S matches "<u>**if** C **then** S</u> foo"
S matches "<u>**if** C **then** S_1 **else** S_2</u>"
S *fails* on "**if** C **else** S"

NB: Note that if we reverse the order of the sub-expressions, then the second sub-expression will never be matched.

Greedy option and repetition

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

$I \leftarrow L^+$
$L \leftarrow$ <u>**a**</u> / <u>**b**</u> / <u>**c**</u> / …

I matches "<u>**foobar**</u>"
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

Comment	←	Begin Internal* End
Internal	←	!End (Comment / Terminal)
Begin	←	**/****
End	←	***/**
Terminal	←	[*any character*]

C matches “**/**ab*/cd**”
C matches “**/**a/**b*/c*/**”
C *fails* on “**/**a/**b*/**

A comment starts with a “begin” marker. Then there must be some internal stuff and an end marker.

The internal stuff must *not* start with an end marker: it may be a nested comment or any terminal (single char).

Formal properties of PEGs

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

What can’t PEGs express directly?

> Ambiguous languages
  - —That’s what CFGs are for!

> Globally disambiguated languages?
  - —$\{\underline{\mathbf{a}},\underline{\mathbf{b}}\}^n$ $\underline{\mathbf{a}}$ $\{\underline{\mathbf{a}},\underline{\mathbf{b}}\}^n$

> 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	←	<u>(</u> Add <u>)</u> / Dec
Dec	←	<u>0</u> / <u>1</u> / … / <u>9</u>

***NB:** This is a <u>scannerless parser</u> — 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);
  }
...
```

Notice how alternative choices are expressed as a series of try/catch blocks. Each rule takes as an argument the current position in the input string. The new position is returned as part of the partial result computed thus far.

NB: Instead of using exceptions, we could encode failure in the Result instances. Then instead of putting alternatives in try/catch blocks, we would have to test each result for failure.

Scannerless parsers are especially useful when mixing languages with different terminals.

Parsing “6*(3+4)”

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

```
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 *
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 )
```

```
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 *
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 )
```

```
[ ... ]
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 )
Eof
312 steps
6*(3+4) -> 42
```

The `SimpleParser` class reports whenever an alternative choice fails, as this will trigger backtracking to try a further alternative.

Here we see that the Prim rule fails initially as its first choice is to look for a parenthesized expression, but instead it finds a digit.

The parse backtracks 13 times and takes a total of 312 steps.

Source code: git://scg.unibe.ch/lectures-cc-examples

Subfolder: cc-SimplePackrat

Memoized parsing: Packrat Parsers

- Formally developed by Birman in 1970s

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);
  }
...
}
```

Introducing a cache in any program is usually straightforward. When you compute a result, first check if you already have a cached value. If so, return it; if not, compute it and save it.

Here we use a hash table to store the results of recognizing a particular non-terminal at a given position in the input. Our packrat parser subclasses the `SimpleParser` class, overrides every method implementing a parse rule with a new one that performs the cache lookup, and defaults to the super method in case there is no cached value.

Memoized parsing "6*(3+4)"

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

```
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 *
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
56 steps
6*(3+4) -> 42
```

A “packrat parser” is a PEG that memoizes (i.e., caches) intermediate parsing results so they do not have to be recomputed while backtracking.

In our grammar this is useful in two places. In the Add rule we may successfully recognize a Mul and then fail on “+ Add”. This would cause the PEG to backtrack and try the second alternative of the Add rule, forcing it to recognize Mul again. With a packrat parser we will see that we already recognized a Mul at position 0 in the input, so we simply retrieve that result instead of recomputing it.

The second case is the Mul rule, which would cause Prim to be parsed again in case the first alternative fails.

What is Packrat Parsing good for?

- Linear cost
  —bounded by size(input) × #(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

Note that we must cache at most # positions for each parser rule.

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

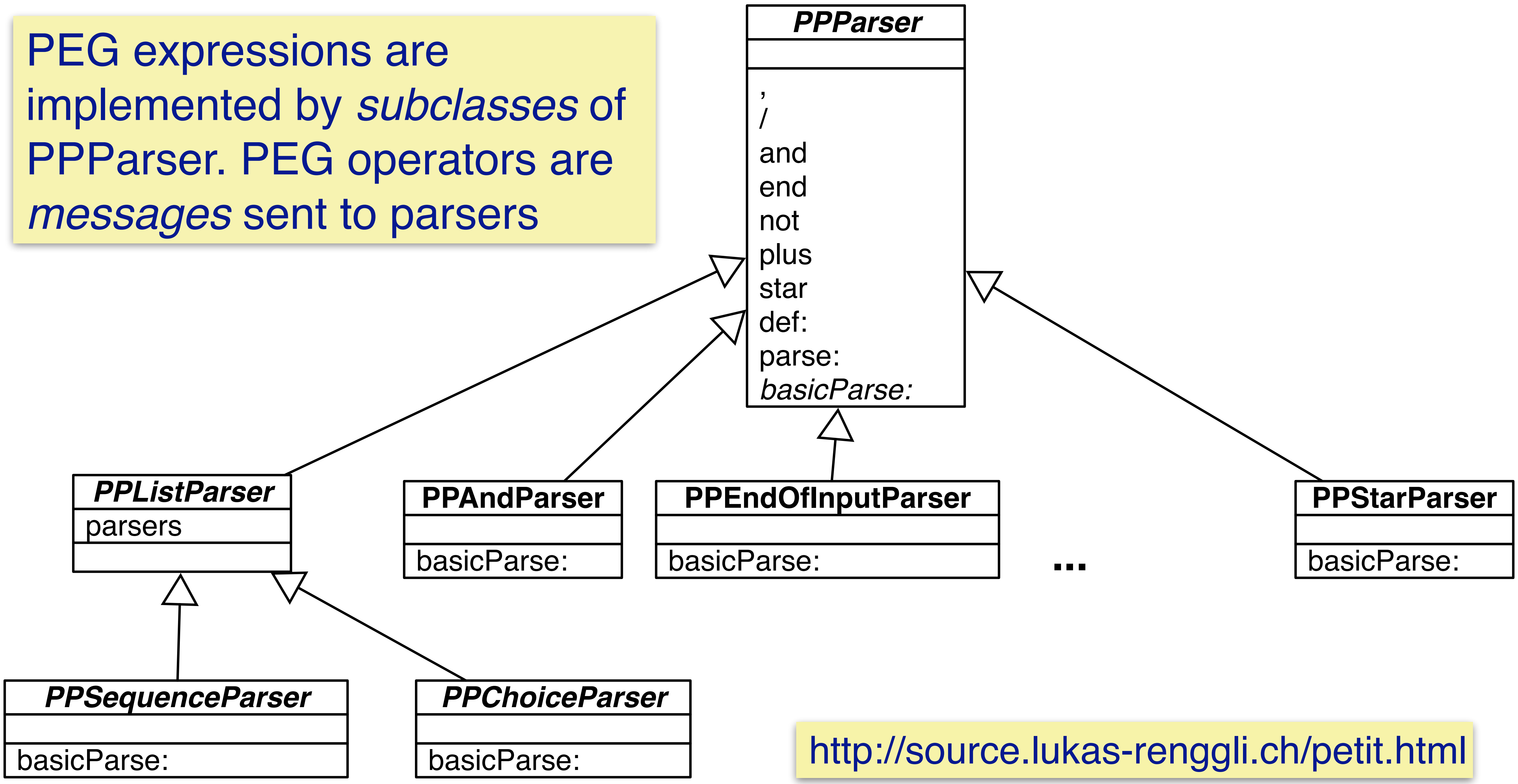
The examples we saw so far implemented PEGs in Java using one method per parser rule.

With parser combinators, each parse rule is a first class value. In functional languages, these values are higher-order functions, which are composed to build more complex parser combinators.

In an OO language, parser combinators are objects. A complex parser is just a tree of objects.

PetitParser — a PEG parser combinator library for 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

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

```
| 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) $)))
```

PetitParser overloads Smalltalk syntax to define a DSL for writing parser combinators.

The dollar sign denotes a character in Smalltalk. To obtain a parser for a character, we send it the message asParser. The comma is used to sequentially compose parsers and the slash creates a prioritized choice.

Semantic actions in PetitParser

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

```
| 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.

goal parse: '6*(3+4)' asParserStream ➔ 42
```

By default, a PP parser just returns a parse tree. In this example, we add semantic actions to parsers. Each action is a block (anonymous function) that takes the parse result and transforms it. The rules here simply evaluate the recognized expressions.

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>()
             .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);
}
```

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?*
- *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?