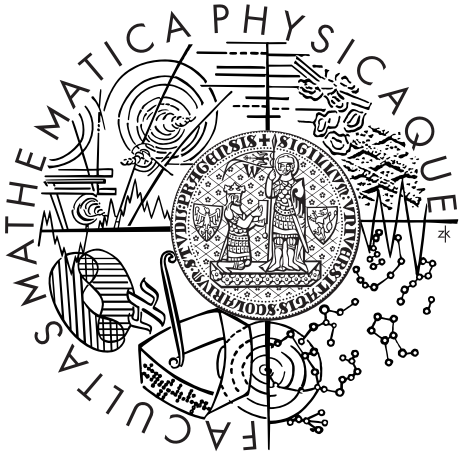
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Ondřej Bílka

Pattern matching in compilers

Department of applied mathematics

Supervisor of the master thesis: Jan Hubička

Study programme: Diskrétní modely a algoritmy

Specialization: Diskrétní modely a algoritmy

Prague 2012

Acknowledgements

I would like to thank to everybody at Charles University who made this possible.

I would like to thank Pavel Klavík for help with the typography.

I would like to thank Andrew Goodall for a help with traps of the English grammar.

I would like to thank my advisor Honza Hubička for help, time, endurance, valuable comments and insights how compilers work in real world.

I would like to thank to many authors on whose work is this thesis built.

I would like to thank my family for their love.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Pattern matching in compilers

Autor: Ondřej Bílka

Katedra: Katedra Aplikované Matematiky

Vedoucí diplomové práce: Jan Hubička, Katedra Aplikované Matematiky

Abstrakt: V této práci vyvineme nástroje na efektivní a flexibilní pattern matching. Představíme specializovaný programovací jazyk amethyst. Jedna z funkcí amethystu je generatování parserů. Také může sloužit jako alternativa k regulárním výrazum. Naš systém umí generovat dynamické parsery. Jejich hlavní uplatnění je tvorba nástroju do IDE jako např. interaktivní zvýrazňovač syntaxe nebo detektor chyb. Amethyst umí zpracovávat i obecné datové struktury. Plánované využití je implementace kompilátorových optimalizací jako například propagace konstant či rozvrhování instrukcí a jiné optimalizace založené na dataflow analyze.

Generované parsery jsou víceméně top-down parsery. Představíme nový algoritmus pro parsovaní strukturovaných gramatik v linearním čase. Amethyst používá techniky z kompilatorů pro optimalizovaní generovaných parserů.

Klíčová slova: packrat parsování, dynamické parsování, strukturované gramatiky, funkcionální programování

Title: Pattern matching in compilers

Author: Ondřej Bílka

Department: Department of Applied Mathematics

Supervisor: Jan Hubička, Department of Applied Mathematics

Abstract:

In this thesis we develop tools for effective and flexible pattern matching. We introduce a new pattern matching system called amethyst. Amethyst is not only a generator of parsers of programming languages, but can also serve as an alternative to tools for matching regular expressions. Our framework also produces dynamic parsers. Its intended use is in the context of IDE (accurate syntax highlighting and error detection on the fly). Amethyst offers pattern matching of general data structures. This makes it a useful tool for implementing compiler optimizations such as constant folding, instruction scheduling, and dataflow analysis in general.

The parsers produced are essentially top-down parsers. Linear time complexity is obtained by introducing the novel notion of structured grammars and regularized regular expressions. Amethyst uses techniques known from compiler optimizations to produce effective parsers.

Keywords: Packrat parsing, dynamic parsing, structured grammars, functional programming

Contents

A bigger picture

This thesis forms the basis of an overarching project: We wish to experiment with language design and optimizations. We seek a language with flexibility beyond Lisp and with good syntax. Our base language is Ruby, which is often described as Lisp with syntax. We strive for high performance. One of our goals is to make a dynamic optimizations and optimizations made on the data structure level possible.

In this work we solve one aspect of this project: A major part of every compiler implementation consists of various forms of pattern matching often written in ad-hoc way. For example tokenization, parsing, expression simplification, dataflow analysis or instruction selection can be seen as instances of pattern matching. We introduce amethyst which is a tool for pattern matching of arbitrary data.

To reach our goals of high effectivity and flexibility we use top-down parsers. For a long time bottom-up parsers were viewed as the only alternative to handle reasonably rich class of languages. However top-down parsers have received a lot of attention recently.

The new formalism of boolean grammars [28] extends context free grammars (introduced by Chomsky in 1956 [8]) to a wider family of languages that includes most of programming languages. A variant of the top-down parser that archives linear time by memoization was introduced by Ford [12]. These parsers can be generated from description in PEG format. We extend this research by introducing notion structured grammars that overcomes several limitations of PEG. We provide a linear time algorithm for parsers of structured grammars that gives exactly the same output as a backtracking top-down parser. The class of languages recognized by PEG is equivalent to the class $\mathrm{REG}^{\mathrm{REG}}$ recognized by amethyst.

Amethyst takes inspiration from an OMeta (2007) [42] which extended parsing expression grammars (2002), which extended regular expressions (1956), [19] which were introduced as a way to describe finite state machines (1943) [22].

One of the extensions made in OMeta is pattern matching of tree-like data structures. We further extend this work in several respects. One is extending pattern matching to arbitrary data structures. In OMeta there are hints of functional language. Amethyst provides several high-level constructs known from functional languages (lambdas, trackable state). A goal is to make grammars made in amethyst more maintainable.

We introduce a framework to make parsing dynamic (Chapter 3), probably the first time this has been done. An editor can add or remove characters and obtains updates to a syntax tree. A dynamic parser recomputes only rules it needs to recompute. For typical workloads a change takes only $O(\log n)$ time. One application will be to make syntax highlighting and other tools easier to write and more accurate.

As a step in experimenting with language design we created a simple dynamic programming language called Peridot.

1. Amethyst language

Amethyst is a pattern matching system.

The purpose of this chapter is to introduce amethyst language and to teach how to use it effectively. We describe amethyst as an evolution of concepts from various pattern matching systems. We will progressively see more hints of functional programming style. In fact amethyst turned out to be full functional language.

Our starting point are regular expressions and we will visit several different settings, each more general than the previous one.

Then we move to the problem of parsing with the focus on top-down parsing. We introduce PEG parser and generalize it to more flexible REG^{REG} parser. (In fact, REG^{REG} stands for relativized regular expressions.) Next we can move into pattern matching of tree-like structures. Finally we generalize our pattern matching to objects that can form arbitrary graphs.

1.1 Notation

For better readability our examples use syntax highlighting. We also use the following notational conventions:

An example code is enclosed in a box like this.
In examples an result of expression is written with the following syntax:

```
2+2 #-> 4
```

Most of the amethyst functionality is implemented by normal amethyst code in standard prologue file. We also show portions of standard prologue in boxes with color like this.

1.2 Technical prerequisites

We assume that the reader knows the basic syntax of Ruby language (we give a brief overview in the next section.) and is familiar with the basics of formal language theory. From Section 1.7 onward we expect an understanding of basic functional programming techniques.

1.2.1 Basics of Ruby

We show several examples of Ruby expressions that we will use in later sections.

Arithmetic offers no surprise:

```
# This is a comment.
# Expected results of expressions are denoted by
# comment #-> result
x=2
y=3
x+y*y #-> 11
```

Function definition and function call are written as follows:

```
def pyth(x,y)
 x*x+y*y
end
pyth(3,4) #-> 25
```

We use the following operations with arrays:

```
[1,2]+[3,4] #-> [1,2,3,4]
x = [1,2]
# We use splat operator to expand arrays
[ x, *x] #-> [[1,2],1,2]
# Splat can be used in function calls
pyth(*x) #-> 5
```

A *closure* is an important concept from functional programing [39]. Both Ruby and amethyst use closures. An example in Ruby follows:

```
def foo(x,y)
  proc{
    x=x+1
    x+y
  }
end
x=1
z=foo(x,2)
z.call     #-> 4
x=0
z.call     #-> 5
```

1.2.2 Getting sources

A source of amethyst can be obtained from git repository by the following command:

`git clone git://github.com/neleai/mthyst.git`

This thesis refers to a version of amethyst that can be obtained by running the following command:

`git checkout thesis`

Examples used in this thesis are also in the `doc` directory of the amethyst.

The peridot language can be obtained by:

`git clone git://github.com/neleai/peridot.git`

Installation and running instructions are in `README` files.

1.2.3 Using amethyst

To use amethyst in a Ruby program you need to load it first:

```
require 'amethyst'
```

Then you can load amethyst source files as in the following example:

```
Amethyst::file 'example1.ame'
```

An amethyst source file is a Ruby source file except for grammar definitions with the following syntax:

```
amethyst Grammar{
  rules
}
```

Hello world program in parser generators is a simple calculator. We follow this tradition too. Constructions used will be the topic of the following chapters.

The source consist amethyst source file `calculator.ame`:

```
amethyst Calculator {
  calculate = add_expr

  add_expr  = add_expr:x "+" mul_expr:y   -> x+y
            | add_expr:x "-" mul_expr:y   -> x-y
            | mul_expr

  mul_expr  = mul_expr:x "*" atom_expr:y  -> x*y
            | mul_expr:x "/" atom_expr:y  -> x/y
            | atom_expr

  atom_expr = "(" add_expr:x ")"          ->  x
            | float
}
puts Calculator.calculate("2-4+2*2--2") #->4
```

and Ruby source file `amethyst.rb`:

```
require 'amethyst'
Amethyst::file 'calculator.ame'
while true
  input = gets
  puts Calculator.calculate(input)
end
```

The file `calculator.rb` is run by the command:

```
ruby calculator.rb
```

For tasks where a simple expression suffices, defining full grammar is not necessary. We can enclose arbitrary amethyst expression `e` in the following construction: `(| e |)`. This creates an object that can be handled in a similar way as a regular expression. So instead of writing:

```
amethyst Hello_World {
  hello = 'hello'
  world = 'world'
  hello_world = hello ' ' world
}
Hello_World.hello_world(s)
```

one can write:

```
(| 'hello world' |) === s
```

1.3 Regular expressions

Regular expressions provide a way to match strings of text and are widely supported by many languages and libraries. They extend search and replace functionality of text editors like `vi`. Typically implementations add nonstandard extensions which we will not consider in this work.

Regular expressions can be formed recursively: An expression can be an atomic expression that can not be decomposed (and typically matches single character) or expressions composed from smaller expressions by some operator.

Atomic expressions

Syntax	Description
`c`	Match character c[1].
`.`	Match arbitrary character.
`[group]`	Match character described in character group.

Operators

`e1e2`	Sequencing
`e1|e2`	Choice
`(e)`	Grouping
`e*`	Iteration: match `e` 0 or more times.
`e+`	Iteration: match `e` 1 or more times.
`e?`	Iteration: match `e` 0 or 1 times.

For example, the expression `[Hh]ello (world|worlds)` matches the strings *"Hello world"*, *"hello world"*, *"Hello worlds"*, *"hello worlds"*.

In Ruby regular expressions are enclosed by "/". We match the example above against the string *"hello world"* by writing:

```
/[Hh]ello (world|worlds)/ === "hello world"
```

Note that the space is also matched literary. This becomes problematic for more complex expressions as they can not be reformatted.

1.4 Amethyst grammars and expressions

The syntax of a regular expression and its equivalent amethyst expression is similar.

We embed amethyst expressions with `(| e |)` syntax. The example from previous section becomes:

```
(| <Hh> 'ello ' ('world'|'worlds') |) === "hello world"
```

[1] Unless c has special meaning in which case you have to escape it.

Amethyst is whitespace insensitive. We need to enclose matched strings with single quotes. The reason why `[]` turns to `<>` will be explained in Section 1.10.

Grammars

Expressions are useful for simple tasks. More complicated tasks are described by *grammars*. Amethyst source file consist of grammars that contain *rules*. Syntax of rule definition and calls is the following:

Pattern	Description
`name = e`	Rule definition
`name`	Rule call

If we want hello world program to be whitespace insensitive we can write it as:

```
amethyst Grammar {
  space  = < \t\r\n>

  hello  = 'hello' space+ 'world'
}
```

As a less trivial example we show rules recognizing integers:[1]

```
amethyst Grammar2 {
  digit  = <0-9>
  int    = '-'? digit+
}

# A rule can be invoked in the following way:
   Grammar.int(       "421") #-> ['4','2','1']
```

Character groups

Character groups provide a concise way to match single character from given set of characters.

Following constructions are supported:

Regular expression	Amethyst	Description
`[a]`	`<a>`	Match character *"a"*
`[aeiou]`	`<aeiou>`	Match any of characters *"aeiou"*
`[a-z]`	`<a-z>`	Match any of characters from *"a"* to *"z"*
`[^abc]`	`<^abc>`	Match any character except *"abc"*
`[[:digit:]]`	`<<digit>>`	Match predefined class

In definitions above characters "`<>\`" have to be escaped.

There are several predefined rules to match POSIX character classes (`alpha`, `alnum`, `digit`, ...). User can also define custom character class, say vowels:

```
vowels = <aeiou>
```

And use it as character group class: `<<vowels>0-9>`.

[1] If we do not care what this rule returns.

1.5 Amethyst expressions

Amethyst consist of a small set of core operators. The rest of amethyst syntax is a syntax sugar that is converted to core operators. Most of amethyst functionality is done by ordinary rules. These rules are contained in file called *standard prologue* in Appendix B. We will show relevant parts of standard prologue as an example.

Explaining exact semantic of core operators will take some time. In this section we only briefly summarize core operator syntax. Various aspects of core operators will be covered later.

Basic operators

Like most pattern matching systems amethyst supports following operators:

Name	Description
`e1 e2`	Sequencing
`e1|e2`	Priorized choice
`(e)`	Grouping
`e*, e+, e?`	Iteration

Lookaheads

When parsing programming languages the decisions which alternative should be used often depend on the future input. This is done by means of *lookaheads*. Due to limited memory of computers in the 1970's the lookaheads were limited to next token. A PEG parser relaxes this restriction by allowing unlimited lookahead. Amethyst also supports unlimited lookaheads but with slightly different syntax:

`e1 & e2`	Positive lookahead
`~e`	Negative lookahead

Positive lookahead is similar to intersection. If input can be matched by `e1` then lookahead matches input by `e2`, otherwise it fails. Negative lookahead succeeds if and only if `e` fails and consumes no input.

In amethyst integers can be recognized by a rule `int`. Based on first character one can decide if integer is positive or negative. We can use positive lookaheads to match positive integers and negative lookaheads to match negative integers in the following way:

```
amethyst Numbers {
  negative_int = ~<0-9>   int
  positive_int =  <1-9> & int
}
```

Atomic expressions

We represent the following atomic expressions as the calling of a rule:

Atomic expression	Rule call	Description
`.`	`anything`	Match single object(character)
`'str'`	`seq("str")`	Match string `str`
`"str"`	`token("str")`	Match string preceded by whitespaces
`<str>`	`regch("str")`	Like `[str]` in regular expressions

Rules `anything`, `seq` and `regch` are implemented as core functionality. Rule `token` is derived and the relevant part of standard prologue follows:

```
_           = < \t\r\n>
token(x)    = _* seq(x)
```

We recommend reading Appendix B containing standard prologue. It is expected that you will not completely understand some parts now[2]. Make a guess what the unknown parts do. This is the best way how to learn a new language and amethyst is no exception.

Enter operator

The operators covered so far deal mainly with matching strings. We need an additional operator `Enter` to deal with general (possibly cyclic) data structures. `Enter` operator is a powerful tool essential to sections 1.10 and 1.11.

Name	Description
`e1[ e2 ]`	Enter operator.

An `Enter` operator matches `e1`. Then it recursively invokes parser to match `e2` on the result of `e1`.

`Enter` operator is one of the most important generalizations of amethyst. It allows us to do pattern matching of object with high level of abstraction which is the topic of Section 1.10 and Section 1.11.

[2]Explaining them is the topic of this chapter.

1.6 External interface

So far we can only decide if an expression matches an input or not. To get useful work done we need integrate amethyst with a programming language. Amethyst is designed to be language independent and the particular language that is used is called the *host language*. In this thesis we use Ruby as the host language.

In amethyst each expression yields a result. Results can be bound to rule-local variables using *variable binding* and processed by host language expressions which we call *semantic actions*. In shortcuts `a` denotes an anonymous variable which does not occur elsewhere.

Functional languages use the notion of referential transparency [38]. Amethyst requires a weaker condition: execution is done in a persistent way. When an alternative fails we revert all changes it made and pretend they never happened.

Semantic actions and variable binding

The syntax of semantic actions and variable binding is the following:

Pattern	Expansion	Description
`{c}`	core	Semantic action.
`-> c newline`	`{c}`	Alternative syntax.
`e:v`	core	Variable binding.

We use Ruby closure support to capture scope as this example shows:

```
(| int:x "+" int:y |).match("2+2")
puts x+y #-> 4
```

Syntax sugar for variable binding

It is common to collect results in an array or do simple conversions. First be expressed by the following syntax sugar:

`e:{ c}`	`e:it {c}`	Do conversion using variable `it`.

For example, imagine that a third party has written a `float` rule. Their API however returns the result as a string. If we want to return a number instead we can write:

```
float2 = float:{it.to_f}
```

When collecting results into array a parameter can be an arbitrary host language expression not just variable:

`e:[ c]`	`e:{c=[*c, it] }`	Append result to array `c`.
`e:[*c]`	`e:{c=[*c,*it] }`	Concatenate result to array `c`.

Semantic predicates

It is possible to test arbitrary properties by *semantic predicates* which are implemented as:

Expression	Expansion	Description
`&{c}`	`core`	Semantic predicate.
`~{c}`	`&{!c}`	Negative semantic predicate.

A semantic predicate expression accepts only if a predicate evaluates to true otherwise it rejects. Otherwise it behaves exactly as the semantic action.

For example, even integers are matched as follows:

```
even = int:x &{ x%2 == 0 } -> x
```

Sometimes we need to represent an expression that always fails. We define the following rule in standard prologue:

```
fails = &{ false }
```

Results of operators

The result of an atomic expressions is typically a string matched by that expression.

Result of operators can be described by the following identities:

Expression	Expansion
`(e1 e2):v`	`e1 e2:v`
`(e1|e2):v`	`e1:v | e2:v`
`(e1&e2):v`	`e1 e2:v`
`(~e):v`	`e`
`(e):v`	`(e:v)`
`e*:v`	`{[]}:a ( e:[a] )* {a}:v`
`e+:v`	`{[]}:a ( e:[a] )+ {a}:v`

Note that lookaheads are always reverted. The main reason is maintainability as lookaheads often cause a rule to be called more times than expected.

Results of rules

Results of rules are passed by an instance variable of parser named `@@_result`.

Expression	Expansion
`name = exp`	`name=exp:@@_result`
`rule:v`	`rule {@@_result}:v`

1.7 Parametrization

As Ruby functions can take parameters so can amethyst rules. The syntax for simple form of parametrization is the following:

Pattern	Description
`name(p1,p2,...) = exp`	Define rule with parameters p1,p2...
`name(p1,p2,...)`	Call rule with parameters p1,p2...

This construction is space sensitive. Placing a space after name is always interpreted as sequencing of rule and expression.

Parametrization in its full generality is more complicated as will be explained in Section 1.13. Here we will give several examples of using parametrization.

Simplest example of parametrization is the following:

```
amethyst Adder {
  add(x,y) = -> x+y
  four = add(2,2)
}
```

Parametrized rule can be called from Ruby with an input string followed by rule parameters:

```
Adder.add("",2,2) #-> 4
```

Several builtin parametrized rules are included in amethyst. We have already seen a parametrized rules `seq` and `token`.

Replacing is text is common task. Say we want replace *"foo"* with *"FOO"* and *"bar"* with *"BAR"*. We can use builtin rule `replace`:

```
Amethyst.replace("fooobars",(| ("foo" | "bar"):{it.upcase} |))
#-> "FOOoBARs"
```

Note that amethyst expressions can be passed also inside grammars.
Example above can be also written as:

```
amethyst Param {
  replace_foobar = replace( (| ("foo" | "bar"):{it.upcase} |) )
}
```

Construction above is called *lambda* [9]. Rule calls inside lambda are resolved lexically [2]. We form a closure for enclosing amethyst rule as is expected from lambda.

Only two parametrized rules are core atomic expressions:

Rule	Description
`apply(x)`	Apply lambda in parameter
`seq(x)`	Match string or apply lambda in parameter

Reason why `apply` does not accept string as a parameter is that we want to do resolving in the callers scope.

Other parametrized rules just use `seq` and `apply`. In standard prologue we follow good practice that rule that accepts string as a parameter accepts lambda too.

As an example consider how rules find and replace are implemented in the standard prologue:

```
find(exp)    = (  seq(exp):x break | .)* .* -> x
replace(exp) = (apply(exp) | .)*:{it.join}
```

Closer look at lambda

Amethyst lambdas form a closure as is illustrated in the following example:

```
amethyst Closure{
  lambda(z) = { (|  {puts z+=1} |) }

  test = lambda(3):x  apply(x) apply(x) apply(x)
}
Closure.test("") #-> 4 5 6
```

Lambda can receive arguments. We can read arguments by calling __ method. This allows implicit syntax for partial application.

```
par(x,y,z) = -> puts(x +y*z )
foo(x)     = -> (| par(__,x,__) |)
```

Example: Parser combinators

Parser combinators [7] are a popular way to implement parsers by people with a functional programming background. They allow the construction of parser expressions by using the host language operators. A combinator support is easy to add by defining operators for amethyst lambda. An implementation follows:

```
amethyst Combinators {
  plus(x,y) = -> (| seq(x)   seq(y) |)
  or(x,y)   = -> (| seq(x) | seq(y) |)
  and(x,y)  = -> (| seq(x) & seq(y) |)
  not(x)    = -> (| ~seq(x)         |)
  star(x)   = -> (| seq(x)*         |)
}

class AmethystLambda
  bin_op=[['+',:plus],['|',:or],['&',:and]]
  un_op =[['star',:star],['~',:not]]
  bin_op.each{|sym,name|
    define_method(sym){|x| Combinators.send(name,nil,self,x)}
  }
  un_op.each{|sym,name|
    define_method(sym){    Combinators.send(name,nil,self)}
  }
end
```

A “hello world” example when we use parser combinators becomes:

```
(|'hello'|) + (|' '|) + (|'world'|)
```

Moreover, a user can write:

```
(|'|'|)|(|'|'|)
```

instead of

```
'|' | '|'
```

1.8 Amethyst extends PEG

Parsing expression grammars (PEG) were introduced by Ford [12]. Our parser started as a PEG parser but evolved into a more general language. In this section we explain the similarities and differences between PEG and our parser.

PEG operators

A typical PEG parser defines expressions formed by the following operators:

Operation	Description
`e1 e2`	Sequencing
`e1|e2`	Ordered choice
`e*`,`e+`,`e?`	Iteration
`&e`,`~e`	Lookaheads

Sequencing and choice

Parsing expression grammars achieve linear time by making choice deterministic and by memoization. In PEG *ordered choice* tries alternatives at left to right order and when an alternative succeeds it does not try further alternatives.

Amethyst extends this choice to *priorized choice* that does backtracking. Linear time is obtained by adding several natural conditions to recursion as is described in Chapter 2.

To describe semantic of our parser we chosen to define auxiliary constructs `Cut` and `Stop` that simplify description [3]. A compiler may use different representation for example one defined in 2.

Our choice operator tries alternatives in left to right order. When an alternative succeeds it does not try further alternatives. We extend choice with `Cut` operator that when encountered it prevents parser form trying other choices. This allows more trackable description of the lookaheads.

Then behavior of operators from the Section 1.5 can be described by the following table:

Operation	Expansion	Description
`e?`	`e|{nil}`	Make `e` optional.
`Cut`	auxiliary	Like ! in prolog
`~e`	`e Cut fails | {nil}`	Negative lookahead.
`e1 & e2`	`~~e1 e2`	Positive lookahead.

[3] similar situation is extending reals to complex numbers.

Iteration

To describe iteration terminating conditions we define an additional atomic expression:

`break`	`core`	To terminate iteration.

We explain iteration by auxiliary *repeat-until* operator. Repeat-until repeatedly tries to match its body and ends only after `Stop` is encountered. If that iteration fails then repeat-until fails.

`e**`	auxiliary	repeat-until
`Stop`	auxiliary	Stop iteration
`e*`	`e**`	When `e` contains `Stop`,
`e*`	`(e|Stop)**`	otherwise.
`break`	`Cut Stop`	Possible expansion.

Examples

Operators `Cut` and `Stop` were introduced to describe a semantic of `break`. A common task is to collect characters until certain character occurs. An `until` rule defined in standard prologue has following implementation:

```
until(chr) = ( seq(chr) break
             | '\\':[x] .:[x]
             | .:[x]
             )* -> x.join
```

For example, rule `line` can be implemented as:

```
newline = '\r\n' | '\r' | '\n'
line    = until(| newline |)
```

Some functionality of `C` standard library can be translated into amethyst as:

C variant	Amethyst variant
`scanf("%i")`	`int`
`scanf("%f")`	`float`
`scanf("%[xyz]")`	`until(| <xyz> |)`
`scanf("%s")`	`until(| _ |)`
`gets`	`line = until(| newline |)`

1.9 Inheritance

In object oriented languages inheritance is a form of reusing code by subtyping existing objects. In Ruby class names must be capitalized. Ruby has simple inheritance with mixins as is shown it the following example:

```
class Foo
  def foo; 42      ;end
end
puts Foo.new.foo #-> 42
class Bar < Foo
  def foo; super+1 ;end
end
puts Bar.new.foo #-> 43
module Baz
  def foo; super*2 ;end
end
class Bar < Foo #class can be defined piecewise.
  include Baz   #include module
end
puts Bar.new.foo #-> 85
#Ruby implements mixin by inserting class between
#current and parent class
```

Inheritance in amethyst

We reuse Ruby class system for inheritance. Ruby class names must be capitalized.

```
amethyst Foo {
  foo = {42 }
}
amethyst_module Mod {
  foo = super:x { 2*x }
}
amethyst_module Baz {
  baz = {"baz"}
}
class Bar < Foo
  include Mod
end
amethyst Bar < Foo {
  foo      = super:x {x+1}
  foo_orig = Foo::foo
  baz      = Baz::baz
}
```

One can use `Grammar::rule` syntax to call rule from ancestor or rule from module. Calling rule of arbitrary grammar is not allowed.

1.10 Pattern matching of tree-like structures

Amethyst takes inspiration from an OMeta (2007) [42] which extended parsing expression grammars (2002). One of extensions made in OMeta is pattern matching of tree-like data structures. We further extend this work in several respects. One described in the next section is extending pattern matching to arbitrary data structures with possible cyclic references.

All operators defined so far carry over into this setting. An `Enter` operator and parametrized rules are essential for this transition.

1.10.1 Pattern matching in functional languages

Most functional languages offer limited form of pattern matching. While syntax is different it usually boils down to the following constructs:

Expression	Description
`Struct`	Match when it is described structure with given name
`:x`	Bind head to variable x
`exp1 exp2`	Sequencing
`exp1 | exp2`	Choice
`[ exp ]`	Enter - take head and match it recursively with exp

Our framework extends these operations with iteration constructs and other features.

1.10.2 Matching nested arrays in amethyst

Recall following amethyst operators. For matching arrays `Enter` can omit leading *"."* as syntax sugar.

Expression	Expansion	Description
`.`	core	Match single element
`e1[ e2 ]`	core	Enter operator.
`[ e ]`	`.[ e ]`	For nested arrays.

Then matching of arrays is quite natural:

```
(| .:x [.:y .:z ] |).match([1,[2,3]])
puts x,y,z #-> 123
```

1.10.3 Classes and pattern matching in Ruby

A common convention to construct tree like structures in Ruby is to define `[]` class method. One way how to construct syntax trees in Ruby source code is the following:

```
Plus[1,Times[2,3,Plus[4,Plus[1,5]]],3]
```

In Ruby membership is tested by case construct. We demonstrate it on contrived implementation of logarithm:

```
def log10(x)
    case x
    when 0          ; raise "not defined"
    when 1..9       ; 1
    when 10..99     ; 2
    when Float      ; log10(x.to_i)
    else            ; log10(x/10)+1
    end
  end
```

A case match is done by invoking `===` method. Use cases of this method are diverse as demonstrated by the following examples:

Left argument	Test performed
`true,false,nil`	Equality.
`42, 3.14`	Equality.
`-42..42`	Range membership.
`Class`	Class membership.
`/exp/`	Regular expression match.

1.10.4 Class membership

Implementation of matching of basic types depends on the host language. We inform amethyst about this by defining a parametrized rule `member`. To implement `member` rule in Ruby we use `===` operator from previous section:

```
member(x) = .:a &{x === a} {a}
```

We define tests for following basic types:

Expression	Expansion
`true,false,nil`	`member(true)`,...
`42`	`member(42)`
`-42..42`	`member(-42..42)`
`Class`	`member(Class)`

1.10.5 Building abstract syntax trees

When we parse we typically build some abstract syntax tree. The following atomic expression makes creation of AST more convenient.

Expression	Expansion	Description
`@Class`	`{Class.create(local_variables)}`	Create object.

This syntax also encourages proper naming of variables. Assume we want to change calculator from first section to produce a syntax tree. Possible implementation is:

```
class Add
  def self.create(hash)
    a=Add.new
    a.x=hash[:x]
    a.y=hash[:y]
    return a
  end
  def amethyst_array
    [@x,@y]
  end
end
# ...
amethyst Calculator_AST {
  calculate_ast = add_expr

  add_expr  = add_expr:x "+" mul_expr:y   @Add
            | add_expr:x "-" mul_expr:y   @Substract
            | mul_expr

  mul_expr  = mul_expr:x "*" atom_expr:y  @Multiply
            | mul_expr:x "/" atom_expr:y  @Divide
            | atom_expr

  atom_expr = "(" add_expr:x ")"        -> x
            | float
}
```

1.10.6 Pattern matching of tree like structures

Amethyst can match any object as an array. First[4] amethyst tries to call method `amethyst_array` and to do matching on returned array. If `amethyst_array` method is not defined it pretends that empty array was returned. This is useful when we match arbitrary objects.

`Enter` operator combined with property testing allows us write evaluator to calculator in the following way:

```
amethyst Evaluator < Calculator_AST {
  eval = Add[      eval:x eval:y ] -> x+y
       | Times[    eval:x eval:y ] -> x*y
       | Multiply[ eval:x eval:y ] -> x*y
       | Numeric

  calculate = calculate_ast=>eval
}

Evaluator.calculate("2+2") #-> 4
```

Example above was created to show possibility of the following simplification:

```
amethyst Evaluator < Calculator_AST {
  eval = Add[                eval:x eval:y ] -> x+y
       | (Times | Multiply)[ eval:x eval:y ] -> x*y
       | Numeric

  calculate = calculate_ast=>eval
}

Evaluator.calculate("2+2") #-> 4
```

If we wanted to also represent addition as a n-ary operation we could extend previous example in the following way:

```
amethyst Evaluator < Calculator_AST{
  eval = plus[                  eval:x eval:y] -> x+y
       | (Times | Multiply)[    eval:x eval:y] -> x*y
       | Numeric

  plus = Add
       | Plus[ .:first .+:rest ] -> Add[first,Plus[*rest]]
       | Plus[ .:first ]         -> first

  calculate = calculate_ast=>eval
}
               #    2 + 3 +   2*2  +  1 + 2*2
Evaluator.eval(Plus[2,3,Multiply[2,2],1,Times[2,2]]) #-> 14

Evaluator.calculate( "2+2" )#-> 4
```

Rule `plus` shows a way to archive an independence of representation. It allows us to freely switch between a representation of addition as an array of summands and a recursive representation.

[4]Unless object is a String or Array

1.11 Matching arbitrary objects

Most of the rules are written to match element of an array. The `Pass` operator that behaves like the `Enter` operator except it wraps the first result into one element array.

Name	Expansion	Description
`e1=>e2`	`e1:a {[a]}[e2]`	Pass operator.

As Ruby is object oriented language you can discover state of object only by method calls. Inside `Enter` operator you can call matched object methods. The syntax is the following:

Atomic expression	Description
`@method`	Call method of matched object.
`@method(a1,a2)`	Call method with arguments.

You can call matched object methods inside semantic acts with same syntax.

Note that disambiguation between method call or object creation is based on the fact that in ruby all class names are capitalized.

Example:

We can also write evaluator by accessing object methods:

```
amethyst Evaluator {
  eval = (Add | Times | Multiply)[ @x=>eval:x @y=>eval:y
                                   -> @is_a?(Add) ? x+y : x*y
                                 ]
       | Numeric
}

Evaluator.eval(Calculator_AST.calculate_ast( "2+2*2" ))#-> 6
```

We can match arbitrary objects, for example hashes.

```
amethyst Match_Hash {
  match = @fetch(:b)=>[ .:x .:y ]
        -> x*y+@fetch(:a)+@fetch(:c)
}
h = {:a=>1,:b=>[2,2],:c=>4}
Match_Hash.match(h) #-> 9
```

1.12 Dataflow analysis generalizes tree traversal

Dataflow analysis [20] is important technique in compiler optimization. It generalizes tree traversal to handle cyclic dependencies.

We illustrate dataflow analysis on real world example that amethyst needs to solve. We first start with two simpler problems where simpler approach is sufficient until we get into a situation where dataflow analysis is necessary.

First example: In regular expressions

We are given a regular expression and want to know a minimal size of string that matches this expression. For simplicity we are given expression as syntax tree consisting only of immutable `Or`, `Seq`, `Char` nodes for binary choice, sequencing and to match character.

```
amethyst Regexp_minimal_size {
  value = Seq[ value:v1 value:v2 ] -> v1+v2
        | Or[  value:v1 value:v2 ] -> min(v1,v2)
        | Char                     -> 1
}

puts Regexp_minimal_size.value( #abc|de
  Or[Seq[Char['a'],Char['b'],Char['c']],
     Seq[Char['d'],Char['e']]])
#-> 2
```

Second example: Adding rules

Now we add rule calls represented as an immutable node `Rule` containing link to body to execute. An example follows:

```
# foo    = 'foo'
# bar    = 'bar'
# foobar = foo bar
foo    = Seq[Char['f'],Char['o'],Char['o']]
bar    = Seq[Char['b'],Char['a'],Char['r']]
foobar = Seq[Rule[foo], Rule[bar]]
```

As far as no recursion is present we can modify our traverser into:

```
amethyst Rules_minimal_size {
  value = Rule[ value:v          ] -> v
        | Seq[ value:v1 value:v2 ] -> v1+v2
        | Or[  value:v1 value:v2 ] -> min(v1,v2)
        | Char                     -> 1
}

Rules_minimal_size.value(foobar) #-> 6
```

And it will always terminate and produce the correct result.

Dealing with recursion

When recursion is present then dataflow analysis becomes necessary.

Dataflow analysis is a method of solving sets of monotonic equations over arbitrary lattice. In our case we use the lattice associated to ordering of natural numbers. We interpret `value` rule in previous example as an inequality that bound a size of expression based on sizes of its subexpressions. We implement the well known worklist algorithm [20] using it to find a minimal solutions of the dataflow equations that correspond to our inequalities.

The algorithm starts with a setting everywhere a value zero. This violates some inequalities. When an inequality is violated we increase left size to value of right side. We repeat this until all inequalities are satisfied. We use algorithm by inheriting from `Dataflow` grammar that is implemented in the next section. Algorithm terminates when all inequalities are satisfied and each `value` attains minimum among all solutions.

We do not have to change our code much to use this analysis:

```
amethyst Rules_minimal_size < Dataflow {
 flow  = Rule[ visit:v           ]  -> v
       | Seq[  visit:v1 visit:v2 ] -> v1+v2
       | Or[   visit:v1 visit:v2 ] -> min(v1,v2)
       | Char                      -> 1
}

class Rules_minimal_size < Dataflow
  def lattice_bottom   ; 0       ; end # Starting solution.
  def lattice_join(x,y); max(x,y); end
end
```

A monotonicity in our case means that if we increase `value` in right side then corresponding value at left side can not decrease. This is in our case true.

Conditions in which this algorithm terminates is finite height of a lattice. This means that for every value we can bound number of increases until we reach this value by same constant. If no recursive rule without terminating condition is present then we know that minimal solution satisfies this condition and our algorithm terminates.

1.12.1 Implementing dataflow analysis

This analysis deals only with immutable objects. It is possible to support mutable objects if you add timestamps to inform if object was changed or not.

In this section we describe a variant of incremental dataflow analysis [34]. We added two new properties.

First property is that analysis is dynamic. User does not have to construct data flow graph. A graph is learned automatically and dependencies vary based for different value assignments. This property can naturally describe concepts like shortcircuit evaluation or flow-sensitive analysis.

Second is that analysis is lazy in sense that it does not compute values until they are necessary to compute.

A simple implementation of our analysis follows.

An amethyst interface is the following:

```
amethyst Dataflow {
  visit =  .:x  {depends(x);@@vals[x]}

  root = .:x analyze(x)

  getvalue(v) = {@@vis=v; v}=>visit
}
```

And a simple analysis based on worklist algorithm follows.

```
class Dataflow < Amethyst
  def value(e)
    @active={}
    @activea=[e]
    while el=@activea.pop
      @active.delete(el)
      @depend.delete_all_edges_to(el)

      val=getvalue(el)
      val=lattice_join(val,@vals[el])
      if val > @vals[el]
        @vals[el]=val
        @depend.edges[el].each{|d| addactive(d)}
      end
    end
    @vals[e]
  end

  def depends(e)
    @depend.add(e,@vis) if !@depend.edges[e].include?(@vis)
    if !@visited[e]
      @visited[e]=true
      addactive(e)
    end
  end

  def addactive(e)
    if !@active[e]
      @active[e]=true
      @activea<<e
    end
  end

  def initialize
    @depend=Oriented_Graph.new
    @vals=Hash.new(lattice_bottom)
    @visited={}
  end
end
```

1.13 Parameters as an object

Now we covered enough background to decribe the full form of amethyst parametrization.

In Ruby we can pass parameters in several ways:

```
def a(x,y)
  x+y
end
puts a(2+2) #->4
def b(x=1,y=2)
  x,y
end
puts b(2)   #->4
def c(x,y,*ary)
  ary.inspect
end
puts c(1,2,3,4,5) #-> [3,4,5]
def with_block
  yield(1) + yield(3)
end
with_block{|x| x+3} # -> 10
# ruby1.9 emulates named parameter by passing last parameter
# as hash and allowing to omit {}.
def named(x)
  puts x.inspect
end
named(:foo=>1,:bar=>2) #-> {:foo=>1,:bar=>2}
```

Amethyst parametrized calls are done by creating special object and pattern matching it againist definition.

We can describe them by the following shortcuts:

Pattern	Description
`name(pattern)=e`	`_name(args) = {args}=>pattern e`
`name(a1,a2,key1:val1)`	`_name(Arguments[[a1,a2],{key1=>val1}])`

We use a convention similar to block passing in Ruby.

`e(c1,c2)(| e2 |)`	`e(c1,c2,(| e2 |))`

In matching arguments we use different syntax. Following syntax express idioms common in argument passing more directly and extends syntax of Ruby argument passing.

`name`	`.:name`	Positional argument
`*name`	`.*:name`	Splat operator
`name:e`	`e:name`	Match amethyst expression
`@name`	`@name:name`	Named argument
`@name:e`	`@name=>e:name`	Match named argument with expression
`name=val`	`(.|{val}):name`	Optional argument
`@name=val`		Optional named argument

Example with use cases follows:

```
amethyst Parametrization{
  opt(x,y=1) = x+y
  use_opt    = opt(1,2) #-> 3
  use_opt2   = opt(1)   #-> 2
  multi(x,*y) = -> y
  use_multi = multi(1,2,3) #->[2,3]

  check(x:String,y:String) = -> x+y
  use_check  = check("a","b") #->"ab"
  use_check2 = check(1,2)
             | -> "failed"  #-> "failed"

  named(@x=1,@y=2) = -> x+y
  use_named  = named(x:3,y:3) #-> 6
  use_named2 = named(x:2)     #-> 4
  use_named3 = named(y:1)     #-> 2
}
```

Example: Syntax highlighting

A syntax highlighting in this thesis was relatively simple to implement by amethyst parser. This example relies on parametrized rules.

Consider the following simplified part of amethyst grammar:

```
postfixed =  term
              ( '=>' term
              | '[' expression "" ']'
              | <+*?>
              | ':' '[' (key | name) ']'
              | ':'     (key | name)
              | inline_host_expr
              )*
```

This grammar can be annotated by colors in the following way:

```
postfixed =  term
               ( color("blue" )('=>')     term:e
               | color("blue" )('[' expression:e "" ']' )
               | color("black")( <+*?> )
               | color("green")( ':' '[' (key | name) ']' )
               | color("green")( ':'     (key | name)     )
               | inline_host_expr
               )*

#a sample implementation of color can be
color(col,lam) = {pos}:oldpos apply(lam):r
                 {color_by(col,oldpos,pos)} -> r
```

This approach prevents any changes to the actual text representation of the input (as opposed to translating abstract syntax trees back to text form). The annotation is straightforward. It is realistic to expect advanced users of IDE to write new grammars if amethyst was used as a syntax highlighting engine. This approach also benefits from a dynamic parsing (Chapter 3).

1.14 Taming state

Purity is important concept in programming languages. We say that function is pure when it can not produce any side effect. Advantage of pure functions is that they are easy to reason about. When function is not pure then its behaviour depends on operations made before that function, also known as *state*. Often we have to add state to function as a necessary evil. We will present several constructions that make state behavior more predictable.

We take inspiration from several earlier attempts. We could view Warth's worlds [42] as first attempt. However as worlds are applied only for position tracking so all work is left to programmer. The rats parser [16] recognizes problem and proposes transaction. Again bookkeeping is left to programmer. In general setting Tanter's contextual values [35] are more general than Warth's worlds [42].

Amethyst uses similar idea. For modality reasons we must split contextual values to two cases: Contextual argument and return.

Local state

Local state refers to how can values of local variables inside function change. Functional languages use notion of referential transparency [38]. We use weaker notion. When an alternative fails the we revert all local variables to a state just before alternative was tried.

Lookaheads are especially dangerous because they break assumptions programmer makes about state we always revert to state before lookahead.

Reverting of local state may come as a surprise in a context of initialing variables:

```
foo = {x=4}  fail | success {puts x}  #-> nil because x=4 was reverted.
foo = {x=4} (fail | success {puts x}) #-> 4
```

This can be done effectively by data structures that do a backtracking persistence.

Global state

Handling global state is more tricky. Memoizing parsers cause objects to be shared unexpectedly. Following example returns a modified object instead of correct unmodified one.

```
foo:x {x.a=4} bar | foo
```

Here backtracking persistence can not help as memoized value would be reverted back to nil.

There are ways how mitigate this problem.

- Blame the programmer.
- Recursively clone everything. When naively done we are about as slow as if we would recalculate everything. Using full persistence can typically reduce overhead to constant factor [11]. Disadvantage is that all user structures must support persistence.
- Recursively make every result immutable. This preserves time complexity as we make every object immutable at most once.

  We chosen the last alternative as it is conceptually simplest alternative. Dynamic parsing benefits from immutability as we will see in Chapter 3.

1.14.1 Contextual arguments and return

Are a more transparent way to model a global state than by global variable.

For supplying context we use contextual argument `@>name`. A contextual argument is accessible to all rules that current rule calls. However a change of contextual argument in son does not change parent's contextual argument. We illustrate this on example:

```
foo1 = {@>name="foo"} foo2          {puts @>name} #-> foo
foo2 = foo3
foo3 = {@>name=@>name+"bar"} foo4 {puts @>name} #-> foobar
foo4 =                              {puts @>name} #-> foobar
```

Second most frequent use of global state is to collect some values that are inconvenient to collect directly.

A contextual return `@<name` is concept dual to contextual arguments and can be viewed as a set such that every parent gets union of contextual returns of his sons. This also elegantly handles case when contextual return does not return anything. Again we illustrate contextual return on example:

```
foo1 = foo2 {puts @<names} #-> ["foo","bar","baz"]
foo2 = foo3
foo3 = foo4 suppress bar
foo4 = { @<names << "foo" }
suppress = sup {@<names=[]}
sup  = { @<names << "suppressed" }
bar  = { @<names << "bar" } baz
baz  = { @<names << "baz" }
```

By defining contextual arguments and returns in this way a memoization respecting global state becomes trackable.

We defer describing how to implement these concepts into Section 1.14

1.15 Maintainance

One of the design goals of amethyst is to allow users write general purpose grammars that can be extended as the described language or protocol evolves.

To get a specification of a language or protocol write:

```
Amethyst::pull 'grammar:version'
```

Which loads given version of grammar, downloading it from central repository if necessary. Grammar obtained in this way is immutable and will be always same on all machines. We expect from grammars in repository to be stable and do not change often.

However we expect that protocols will evolve. We want to make updating easier by migrations. A proposed command is:

```
amethyst_migrate file grammar:newversion
```

Which will replay refactorings (for example renaming a rule) described in migration files to new version. This could not be always possible[5] in this case we ask programmer to do migration manually.

Migration from regular expressions

We also want to make transition from other framework easier. As a simple example we implemented an functor that convert subset of regular expressions into amethyst expressions. Usage is the following:

```
regexp= /[Hh]ello (world|worlds)/
reg2ame(regexp).inspect #-> (| <Hh> 'ello ' ('world'|'worlds') |)
reg2ame(regexp) === "hello world" #-> true
```

1.16 Error handling

An error detection is important topic on its own. We implemented only a simple strategy that detects misplaced parethness and suggest probable causes. This is a type of problem that that needs global error recovery. It can be formulated as problem that a given sequence of parentheses what is minimal number of parentheses we have to change to get properly parenthised expression. A simple strategy to guess most probable places can be found in files `amethyst/error_recovery.ame` and `lib/repair_errors.rb`.

Position tracking

For position tracking our approach is simple. We subclass string to the class `Origin_Tracking_String`. Information about position automatically propagates through parser and subsequent pipeline. This also allows to wrap and recursively parse substrings while preserving position information.

[5] Fox example code that evaluates strings from standard input.

1.17 Example: Parser of amethyst

We conclude this chapter by explaining amethyst in terms of itself by providing amethyst parser in amethyst. Summary of constructions used is in Appendix A. We omit several parts that are too technical.

Our first task is parse rule and variable names.

```
name           = (<_a-zA-Z> <_a-zA-Z0-9>*)[]:{it.join}
className      = (    <A-Z> <_a-zA-Z0-9>*)[]:{it.join}
```

File structure

Amethyst file consist of grammars and host language code. We make "`amethyst`" a keyword otherwise grammar with error would be interpreted verbatim.

```
file     =( grammar
          | lambda
          | ~('amethyst' _) . )*
```

Grammars

Amethyst grammar consist from rules.

```
grammar = 'amethyst' "" name ("<" "" name | {"Amethyst"} ):parent
          "{" rule*:rules "}"  @Grammar
```

We specify optional parts of grammar by a "?" operator. When we also want to supply default value we use an idiom `("<" "" name | {"Amethyst"} )`.

Rules

```
argsOpt = args('(',')') | {[]}
#For now you can imagine that args('(',')') matches properly
#nested parentheses.

rule = "" name:name ~_ argsOpt:args "=" expression:body @Rule

call = className:klas '::' name:name ~_ argsOpt:arg #Foreign rule
     |                     name:name ~_ argsOpt:arg -> Apply[name,arg]
```

When we want to tell an user reading grammars that whitespaces are forbidden at certain place we use an `~_` idiom even if it is not neccessary for parser.

Sequencing and choice

Sequencing and choice have the usual precedence. At choice we need to forbid interpreting end of lambda as choice. Whitespaces separate sequence elements. We use negative lookahead `~rule_head` to separate rules.

```
expression = listOf((|sequence|),(|"|" ~')'|)):ary @Or_AST

sequence   =  (~rule_head lookaheads)*:ary  @Seq_AST

rule_head  = "" name:name ~_ argsOpt:args "="
```

Lookaheads

Negative and positive lookaheads are recognized by the following expressions:

```
lookaheads = "" neg_lahead:[s] ("&" ~"{" "" neg_lahead:[s])*

neg_lahead = '~' ~"{" neg_lahead:m    -> Lookahead[m,true]
           |  <&~>:n ~_ inline_host_expr:e -> Pred[e,n=='&']
           | postfixed
```

Postfixes

Note that postfixes are left-associative. In particular `a=>b?` is equivalent to `(a=>b)?`.

```
postfixed      = term:from
               (  <*+?>
               | '['  expression:e "]" -> Enter[from,e]
               | '=>' expression:e     ->  Pass[from,e]
               | ':' '[' name ']' | ':' '"' name '"' | ':' name
               | ':' inline_host_expr:{Seq[Bind["it",from] , Act[it]]}
               ):from  )* -> from
```

Atomic expressions

Various atomic expressions are handled by the following rules:

```
cases =  className:klas  ~'::'       -> Apply["clas",klas]
      | (number ('...'|'..') number
        | number)[]:num              -> Apply["member",num.join]
      | '<'  until('>' ):s           -> Apply["regch","/["+s+"]/"]

key   = '(|' expression:exp '|)'     -> Lambda[exp]
      | '@'  className                # Technical
      | '@'  name:name argsOpt:arg   -> Key[name,arg]
      | '@@' name:name               -> Global[name]
```

Semantic actions

Recognizing semantic actions is dependent on the host language. We do not have to understand whole grammar, recognizing pairing tags suffices[6]. Implementation specific to Ruby follows:

```
args(o,c) = seq(o) hostarg*:r seq(c) -> r

hostarg  = key
         | args('(',')')   | args('[',']')    | args('{','}')
         |  '\'' until('\'')
         | '"' interpolated
         | '#' line
         | <^``'"()[]{}>


interpolated  = ( '"' break
                | args('#{','}')
                | '\\'? .)*

inline_host_expr = args('{','}')
host_expr        = inline_host_expr
                 | '->' line:s {"{"+s+"}"}=>[ inline_host_expr ]
```

Note that rule args as example of parametrized application. Also note how we in `host_expr` we parse recursively.

Now we can put everything together to form term:

```
term  = cases
      | call
      | key:{Act[it]}
      | host_expr
      | '.'                      -> Apply["anything"]
      | '[' expression:e "]"     -> Enter[Apply["anything"],e]
      | '"'  until('"' ):s       -> Apply["token" ,quote(s)]
      | '\'' until('\''):s       -> Apply["seq"   ,quote(s)]
      | '#'  line:s              -> Comment[s]
      | '(' expression:x ")[]" {x}=>collect
      | '(' expression:x ")"     -> x
```

[6]We use more complicated rules to recognize local variables

2. Implementation

In this chapter we describe main techniques used in amethyst parser generator. We introduce novel notion of structured grammars and formalism of relativized regular expressions that enables us to produce effective top-down parsers for wide family of languages.

A top-down parsing implementation can be viewed as bunch of mutually recursive functions recognizing individual rules in grammar description. Top-down parsers are easy to implement and furn fast for simple grammars.

But naively implemented parser of the following rule:
`R="aa" R | "a" R`
on *"aaaaaaaaaaaaaaaaaaa..."* can take exponential time.

Incorporating left recursion also causes problems. A naive parser of
`L=L a`
would call L infinitely many times.

In natural language processing we typically want to enumerate possible interpretations of ambiguous grammar.

Frost [13] gave $O(n^4)$ algorithm that outputs compact representation of all parses [36] and handles left recursion as recursive descend. Parsing expression grammars allow unlimited lookahead. Okhotin [29] suggest to extend context free grammars with lookahead to class of *boolean grammars*. Again his algorithm for boolean grammars had complexity $O(n^4)$. Both these algorithms were improved by variant of Valiant algorithm [41] to obtain complexity $O(M(n) \log n)$ where $M(n)$ is time of matrix multiplication. When boolean grammars are restricted to *unambiguous boolean grammars* there exists $O(n^2)$ algorithm.

For programming languages ambiguity is undesirable. One of approaches are parsing expression grammars defined by Ford [12]. A parsing expression grammars (PEG for short) can be viewed as a top-down parser that places three additional constraints. First is that rules are deterministic. Second is restricting choice operator `|` to *ordered choice* operator `/`. Once an alternative of ordered choice succeeds then choice succeeds then we do not backtrack if something fails later. Third is that iteration is greedy and does not backtrack.

This definition without backtracking introduced problem of *prefix hiding*, an expression `"a"/"ab"` does not match string *"ab"*.

Seaton in his Katahdin language [33] uses different *longest choice* operator to partially solve this problem. The longest choice tries all alternatives and deterministically chooses the longest match. However this does not eliminate the prefix hiding completely. Parser of:
`"␣"* "␣foo"` (`*` is iteration operator) still does not match *"␣foo"*.

We take another approach. Programming languages use only two types of recursion: iteration and nested recursion. By making this information explicit we can generate linear time parsers that are equivalent to the fully backtracking ones.

We present new formalism of relativized regular expressions REG^{REG}. Our formalism relaxes determinism of PEG grammars. As in PEG we support arbitrary lookaheads. Previous results can be easily derived using REG^{REG} formalism.

Although REG^{REG} seems stronger than PEG we show that PEG and REG^{REG} are equivalent.

2.1 Structured grammars

We devise approach to describe programming languages which we call *structured grammars*. We build on an analogy with structured programming languages.

As programs used arbitrary *goto* constructs, grammars use arbitrary forms of recursion. To make programs more readable, programming languages was extended by adding structured control flow constructs making it easier for developers to read the code on a local basis without spending hours to understand the whole context. We seek similar goals with introduction of structured grammars.

Assume we are given a grammar for the fully-backtracking top-down parser. We say it is *structured grammar* if it satisfies the following conditions:

1. *Transparency of semantic actions.* We can imagine that parser is augmented by an oracle that may decide that alternative will eventually fail. The parser should display same output regardless if we tried that alternative and failed or used the hint from the oracle. Lookaheads form important case. We always revert actions made by lookaheads.

2. Recursion is restricted to *iterative* and *nested recursion*.

   (a) Iterative: For example, arguments of function in C are lists of `expressions` separated by ",". We typically use iteration `*` operator. Iteration can be also described by left recursive or by right recursive rules. When possible iteration should be written in way that is associative.

   (b) Nested recursion: What is not iteration can be described by ***`start`*** and ***`end`*** delimiters. We require user to annotate this concept by operator `nested(`***`start,middle,end`***`)`.

   Simplest example are properly parenthised expressions. They can be described as:
   ```
   exp = nested('(',(| exp |),')')
   ```
   We show two less trivial examples in structured grammar formalism. A while loop in `C` is matched by:
   ```
   'while' exp nested('{',(| stmts |),'}')
   ```
   Python uses indentation to describe nesting. We use a semantic predicate to find where we end. We match python while loops in amethyst as:
   ```
   nested((|   '\n' ' '*:x 'while' exp |),(| stmts |),
          (| &('\n' ' '*:y &{x.size>y.size}) |))
   ```

A nesting should satisfy three natural conditions.

2.1. Position of *`end`* delimiter is determined by position *`start`* delimiter.

2.2. When `nested` starts in smaller position it should end in strictly larger position.

2.3. When both `nested(`*`start,mid1,end`*`)` and `nested(`*`start,mid2,end`*`)` match string then their end positions should agree.

Note that programming languages implicitly follow this convention. Other types of recursion are undesirable because user can not reason about them locally.

One of reasons is that programming languages were described as deterministic context free grammars. Thus they can be written by deterministic push-down automaton. We can model push/pop pair by calling `nested`. Indeed if we did not include lookaheads our class would be equivalent to class of deterministic context free grammars.

Structured grammars offer additional advantages. For example, we can use the structure information to semiautomatically construct error correction tool.

For equivalence with top-down parser our parsing algorithm needs condition 2.1. Without condition 2.2 a parser would be quadratic instead linear time. Condition 2.3 is design guideline which is not needed in our algorithm.

2.2 PEG and REG^{REG} operators

In this chapter we use PEG operators as originally defined by Ford [12].

`'s'`	Match string.
`r`	Rule application.
`e1 e2`	Sequencing.
`e1/e2`	Ordered choice.
`e*  e+`	Iteration.
`&e  ~e`	Positive and negative lookahead.
`{a} &{a}`	Semantic action and predicate.

We relax determinism of PEG to REG^{REG} expressions. We can describe every structured grammar by REG^{REG} rules with linear time guarantee. A REG^{REG} expressions mostly use the same operators as PEG. Difference is that operators do backtracking except of `nested` which behaves deterministically.

`nested(start,mid,end)`	Nested operator.
`e1|e2`	Priorized choice.
`e* e+`	Backtracking iteration.
`e1[e2]`	Enter operator.

2.2.1 Simple algorithm

We will describe our parser in functional programming style pseudocode in continuation passing style [15]. We denote lambda as:
`\lambda(arguments){body}` and call it with `call` method.

We start with simple implementation and will progressively add more details.

A REG^{REG} parser behaves mostly as a top-down parser. We use the function `match(e,s,cont)` where `e` is expression we match, `s` is current position and `cont` is a continuation [31] represented as lambda.

```
match( r  ,s,cont) = match(body(r),s,cont)
match('c' ,s,cont) = if s.head=='c'              ;cont.call(s.tail)
                     else                        ;fail

match(e f ,s,cont) = match(e,s,\(s2){ match(f,s2,cont) }
match(e|f ,s,cont) = if match(e,s,cont)          ;success
                     else                        ;match(f,s,cont)

match(~e  ,s,cont) = if match(e,s,\(s2){success});fail
                     else                        ;cont.call(s)
match(e*  ,s,cont) =
  cont2 <-    \(s2){ if match(e,s2,cont2)       ;success
                     else                        ;cont.call(s2)
                   }
  cont2.call
```

Pseudocode above describe naive top-down parser. For REG^{REG} class we restrict recursion and add `nested` operator:

```
match(nested(st,mi,en) ,s,cont) =
  s3 <- match((st mi en),s,\(s2){success})
  if s3    ; cont.call(s3)
  else     ; fail
```

2.2.2 Equivalence with top-down parsers and PEG

We prove that for structured grammars REG^{REG} parser finds same derivation as fully backtracking one. As top-down parser does not directly support left recursion we do not consider left recursion in this section.

An implementation of the fully backtracking parser is same as the implementation of REG^{REG} parser in Section 2.2.1 except of `nested`:

```
match(nested(st,mi,en), s, cont) = match((st mi en), s, cont)
```

For sake of proof we transform rewrite implementation of `nested` in REG^{REG}parser to equivalent one. In `nested` we only consider the first alternative in the way the following pseudocode suggests:

```
match(nested(st,mi,en), s, cont) = first <- true
  match(s, (st mi en), \(s2){
    if first ; first <- false
             ; cont.call(s2)
    else     ; fail
  })
```

An equivalence with top-down parser can be proved by easy induction on the nesting level.

1. When expression contain no nesting we have identical implementation.
2. Assume we proved proposition for nesting level $\ell - 1$. We prove level ℓ by second induction on the number of `nested` calls in the continuation of level $\ell - 1$.
   (a) For continuation that does not call `nested` we use same argument as in 1.
   (b) Assume we have continuation that calls `nested` n times. Consider first time we call `nested`. If this call fails it, by induction, also fails in the fully backtracking parser and we are done.

   Otherwise REG^{REG} and the fully backtracking parser first try lexicographically smallest alternative in the recursion tree. If a continuation succeeds a derivation is same by induction.

   If a continuation fails we use assumption 2.1. of structured grammars. Our parser does not try alternatives further. A backtracking parser enumerates all derivations. As every derivation ends in same position and continuation will always fail. Thus the backtracking parser behaves like REG^{REG} parser.

Like not every C program is structured program not every $\mathrm{REG}^{\mathrm{REG}}$ grammar is structured one. We can use `nested` with empty ***`start`*** and ***`end`*** to implement PEG operators. This gives us inclusion PEG $\subseteq \mathrm{REG}^{\mathrm{REG}}$. An opposite inclusion is true but not very enlightening. As there are only finitely many pairs $(e, cont)$ we can for each pair write a PEG rule that emulates $\mathrm{REG}^{\mathrm{REG}}$ algorithm.

For linear time guarantee we still require every recursion except left and right recursion to be annotated by `nested`.

2.3 Relativized regular machines

To better understand languages recognized by relativized regular expressions we introduce the *relativized regular machines* that are similar to nondeterministic finite state machines [30]. We use this formalism as an inspiration for effective low-level implementation of parsers.

It is easy to see that a continuation corresponds to syntactic right congruence class. We use representation that unifies identical expressions and continuations. This can be viewed as NFA state minimization[1].

A *relativized regular machine* is similar to nondeterministic finite state machine. A machine can be described by triple $M = (S, t, a)$ where

S is set of states,
$t : (S, N, S) \to (M, S)$ set of transitions and
$a \subset S$ a set of accepting states.

We have elementary machines that match single character.

Transitions from state s are done in the following way. We put $(M_i, s_i) = t((s, i, r_{i-1}))$ then recursively call machine M_i and if it succeeds we move to its end position and set state to s_i. Based of accepting state r_i this choice reaches we choose a next choice.

[1] NFA minimization is NP-hard in general case. Our approach is a good heuristic.

2.4 Effective implementation

An implementation above runs in linear time but constant factor is quite high. For better constant factor our parser generator applies various optimizations. We use a low-level representation that is suitable for these optimizations.

In this section we describe parser that does not consider semantic actions. Semantic actions will be added in the next section.

Representation of expressions is similar to syntax tree. We use similar technique as compact representation of derivations in Tomita algorithm [36]:

1. All nodes are immutable.
2. We represent all identical subtrees by single object. When we are asked to construct a node optimizer first tries to simplify node by algebraic identities. If after simplification we obtain node identical to previously constructed node we return previously constructed node.

We will again use function `match(e,Args[ ... ] ) -> Result[ ... ]`.
We will extend several times what `Args` and `Result` objects contain. Initially we define the following fields:

`Args.s`	is starting position of string,
`Result.s`	is end position of string,
`Args.cont`	is a continuation.

Objects `Args` and `Result` have method `change` that creates new object with appropriately changed fields.

2.4.1 Sequencing

We represent sequencing operator `head tail` by object with pattern `Seq[head tail]`. Representing sequencing in this way allows `tail` parts to be shared. Implementation is straightforward.

```
match( Seq[head tail],a) = match(head,a.change(
  cont:\(a2){
    match(tail,a.change(cont: a.cont))
  }))
```

2.4.2 Choice and lookaheads

Inspired by relativized regular machines we model choice and lookaheads by more general `Switch` operator. First we need add field `Result.state`. This state will be used to pass information from rules to the `Switch` operator.

The `Switch` operator satisfies the following pattern:
`Switch[ head alt:{state=>tail} merge ]`
`Switch` operator first matches a head. Then it looks what end state head reached and matches tail entry corresponding to that state. Finally it computes final state from states of children by merge method.

For simplicity in this paper we use only two states `success` and `fail`. We use identity function as a merge method. We also add `success` and `fail` rules with obvious implementation:

```
match(Rule[success]) = Result[state: success]
match(Rule[fail   ]) = Result[state: fail   ]
```

This is quite general operator and we illustrate its uses on several examples.
The choice operator backtracks until success state was reached. An implementation is:

```
e1|e2  ->    Switch[ e1 {success: success
                         fail: e2}]
```

Lookaheads can be modeled like:

```
~e   ->    Switch[ Seq[e success ]
                  {success:   fail,
                   fail:      empty} ]
&e   ->    Switch[ Seq[e success ]
                  {success:   empty,
                   fail:      fail } ]
```

A `Switch` makes optimizations easy. Switches can be easily composed. To compose switches `A` and `B` a simplest way is to use states that are pairs (state from `A`,state from `B`). We need to define merge method to compute final state. We can represent these pairs compactly as bit vector. Another optimization is predication. When we know first character we can simplify expression:

```
Switch[ Result[ first_character ]
                  { 'a': expressions that can start by a,
                    'b': expressions that can start by b,
                    ...
```

For choice `e1|e2` we can, based on the result of the partial match of e1, simplify matching of `e2`. For example, consider expression:

```
  (a|b) c (d|f)
| (b|c) c f
```

on string *"bcd"*.
When first alternative matches *"d"* then we know that second alternative will not match. Last choice could pass state to inform first choice about this condition.

An implementation of `Switch` is the following. We hide technical details to merge method. For details see our implementation [5].

```
match_memo(e,a) =
if memo[e,a]; memo[e,a]
else        ; memo[e,a] <- match(e,a)

match(Switch[ head alt merge ],a) =
  r  <- match_memo(head,a)
  r2 <- match_memo(alt[r.state],a)
  merge(r,r2)
```

2.4.3 Iteration

We use low-level repeat-until and `Stop` operators from previous chapter to represent iteration.

Repeat-until can terminate if and only if we encountered corresponding `Stop` in current iteration. We add `stops` field to `Args` to collect encountered stops.

This allows to describe normal iteration `e*` and eager iteration `e*?` as `(e|Stop)**` and `(Stop|e)**` respectively. Repeat-until is equivalent to right-recursion. For example, we can flip between rules

```
R =  a R | b      | c R | d
```

and

```
R = (a   | b Stop | c   | d Stop)*.
```

Except of stop condition the implementation is nearly identical to implementation of `*` operator from Section 2.2.1.

```
match(Stop[st]   ,a) = a.cont.call(  a.change(stops: a.stops+st))
match(Many[st e] ,a) =
  cont2 <- \(a2){
    if a2.stops & st ; a.cont.call( a2.change(stops:a2.stops-st))
    else             ; match(e,a2.change(cont:cont2 ))
  }
  cont2.call(a)
```

2.4.4 Rule call

Rule call only affects scope of variables. When no semantic actions are present we can directly move expression to separate rule and back.

```
match(Rule[ e ], a) = match(e ,a)
```

For `nested` we use similar implementation as before.

```
  match(Nested[st mi en],a) =
    r = match_memo(Seq[st mi en],Args[s:a.s,cont:\(m){success}])
    if r.state==success ; a.cont.call(a.change(s:r.s))
    else                ; fail
```

2.5 Semantic actions

We add semantic actions as was explained in the previous chapter.

While semantic actions are easy to add they complicate other parts of the parser.

We add the following fields:
`Args.closure` closure for semantic actions.
`Args.returned` the result of last expression.
`Result.returned` returned result.

We model semantic act as a function that modifies arguments. For simplicity we model variable binding by semantic act.

```
match( Act[ f ] ,a ) = a2 <- f.call(a.closure)
  a.cont.call(a.change(a2))
```

Now we are ready to add enter operator.

```
match( Enter[e1 e2], a) =
  match(e1,a.change(cont: \(a2){
     match(a2.change(s:a2.returned),cont:\(a3){
       a.cont.call(a3.change(s:a.s))
     }
  }
```

Semantic actions in rule invocation have shared scope. We use closure object to achieve this. A rule invocation becomes:

```
match( Rule[ e ] ,a ) = match(e,
                              a.change(closure:new_closure,
        cont:\(a2){ a.cont.call(a.change(s:a.s,
                                 returned:a.returned))}
    )
```

We also add semantic predicates from the previous chapter This complicates memoization and we, for simplicity, disable memoization when semantic predicate is present.

Support parametrized rules and lambdas is bit technical to add. For parametrized rule we first model arguments by semantic act bound to argument variables. Then we add field consisting of pairs (argument variable,parameter variable) and we initialize new closure according to pairs. For lambda we bind (expression,closure) pair to corresponding variable. We disable memoization when parametrized rule is present for same reasons as with *semantic predicate*.

Memoization becomes more technical. A simplest way how to get linear time complexity is to use two pass parser which in first pass run parser from Section 2.4 and second time we just constructs parse tree. We refine this idea and run both phases in parallel. We use functor `forget_semantic_actions`:

```
match_memo_state(e,a) =
  if (has_predicate(e) | has_predicate(a)) ; match(e,a)
  else ; e2 <- forget_semantic_actions(e)
         a2 <- forget_semantic_actions(a)
         if    memo[e2,a2]  ; memo[e2,a2]
         else               ; memo[e2,a2] <- match(e,a)
```

A simple implementation of `Switch` can be:

```
match(Switch[ head alts merge ], a)
  r  <- match_memo_state(head,a)
  r2 <- match_memo_state(alts[r.state],a)
  if r2.state==fail
    fail(r2)
  else merge(match(head,a),
             match(alts[r.state],a))
```

Sometimes `Switch` knows that the result is not needed. Then we can directly call expression simplified by `forget_semantic_action`. This always happens for lookaheads.

2.5.1 Time complexity

Ford [12] rewrites iteration to recursion for linear time complexity. However most implementations naively use a loop.

It is possible to construct test cases where arbitrary (say k) number of loops are nested together and each fails at the end of input. This leads to time complexity at least n^k for arbitrary k. This can be seen on the following expression:

```
( ( ( ( 'a' )* 'b'
    / 'a' )* 'c'
  / 'a' )* 'd'
/ 'a' )* 'e'
```

on *"aaaaaaaaaaaaaaaaaaaaaa..."*

We memoize continuations precisely for this reason.

For parser from Section 2.4 there are only finitely many expressions and continuations. Thus there are only $O(n)$ memoization pairs `(e,a)`.

With semantic actions we sometimes need to recalculate the result. For a given pair (`nested,position`) we need to recalculate result of every `(e,a)` pair at most once. For general $\mathrm{REG}^{\mathrm{REG}}$ expressions time complexity $O(n^2)$ follows.

For structured grammars this behavior can not happen. We do not have to recalculate when the result state is `fail` or we match in lookaheads. What is left is that we could have two invocations of same `nested` expression with two different positions that recalculates same `(e,a)` pair. But this would mean that both invocations will be accepted with same end position which is in contradiction with condition 2.2 of structured grammars. Consequently the parser of structured grammars runs in linear time

With semantic predicates we can not give any complexity guarantee. To integrate them correctly we disable memoization when continuation contains semantic predicate.

2.6 Memory consumption of $\mathrm{REG}^{\mathrm{REG}}$ parsers

Mizushima et al [25] propose way to decrease the memory usage. We describe similar but simpler approach.

The parser implementation maintain set of live branches in a list `live`. The list is maintained in the following way:

- When parser descends into choice operator then its branches are added to `live` list.
- When parser descends into branch, then it is removed from `live` list.

- When parser encounters cut then branches that were cut are removed from `live` list.

When `live` list is empty we know that subsequent parsing can not return to position smaller than current. We can safely delete all `memo` entries with smaller position. One can observe that `live` list is not needed. The implementation can be further simplified by only keeping track of the size of the list in a counter `alternatives`.

The parser then deletes stale entries from memo table lazily. It keeps track of the rightmost position where `alternatives` was zero. At a time table expansion is needed, all earlier entries are deleted. This avoids the need for the expansion if the table after deletion is at most half full.

Note that if we want to incorporate destructive semantic actions we can in same way defer their evaluation until `alternatives` is zero.

For practical grammars this extension gives nearly constant memory usage. However we can construct examples where this approach does not help, for example in expression:

```
exp* 'x' | exp* 'y'
```

we need to keep memoization entries until end is reached.

Memoization in general setting

The memoization is viewed as alternative to dynamic programming. A naive memoization can have big memory consumption. We show that with few simple trick we can obtain better performance with order of magnitude smaller memory usage. Memoization strategy and automatic memoization were unsurprisingly developed in context of context-free parsing [27]. We use memoization strategy that applies to reducing memoization memory consumption in general.

First step to reduce memory usage is to save values for parameters in the hash table. This gives memory consumption proportional to number of saved values.

When we do not ask question: "What functions must be memoized?" but right one "What parameters must be memoized?" then solution is surprisingly simple: We have only memoize those parameters that took at least say 512 cycles to compute.

There are three additional improvements:

- We can count only cycles that were not memoized by son rule.
- We keep additional small (say 512 element) directly mapped write-back cache. We use this when we notice that values before our threshold are typically rarely reused after say 100 steps.
- Some functions never reach our threshold so we can use separate table to save unnecessary lookups. We are more worried that these lookups thrash cache than direct cost.

This has same asymptotic time complexity as when we memoize everything because rule is memoized at most once and when we do not memoize time complexity is constant.

There is technical problem how measure number of cycles. While x86 offers timestamp counter and core2 it takes about 24 cycles. We currently use simple estimate by counting number of functions that we called as we typically have constant overhead. Alternative way is first run a version that gathers profiling data and then use estimates from that version.

We illustrate our ideas on the following memoized version of Fibonacci numbers in `C` language:

```
typedef struct {long time;long saved;long result;} time_struct;
time_struct timestamp;
struct{int key;long value;} cache[512];
long  memo[1000000];
time_struct memoize_start(int key){
  if (cache[key%512].key==key){
    timestamp.result=cache[key%512].value;
    return timestamp;
  } else if (memo[key]) {
    timestamp.result=memo[key];
    return timestamp;
  } else {
    timestamp.result=0;
    return timestamp;
  }
}
void memoize_ended(time_struct started,int key,long value){
  long time  = timestamp.time -started.time;
  long saved = timestamp.saved-started.saved;
  long time_self = time - saved;
  cache[key%512].key=key; cache[key%512].value=value;
  if (time_self > 128){
    memo[key] = value;
    timestamp.saved += time_self;
  }
}
long fib(int n){
  time_struct started=memoize_start(n);
  timestamp.time++;
  if (started.result) return started.result;

  if (n<2) return 1;
  result=fib(n-1)+fib(n-2);

  memoize_ended(started,n,result);
  return result;
}
```

2.7 From $\mathrm{REG}^{\mathrm{REG}}$ back to REG

We establish a *reg* functor. We use it to analyze $\mathrm{REG}^{\mathrm{REG}}$ expressions.

A *reg* functor assigns to each relativized regular expression e a regular expression $reg(e)$. A $reg(e)$ satisfies approximation condition that if e accepts s then $reg(e)$ accepts s but converse is not necessary true.

We can extract useful information testing if the intersection with a suitable regular language is empty.

```
empty(e)           = reg(e    ) ∩ reg( ''   )
first_char('c',e)  = reg(e    ) ∩ reg('c' .*)
overlap(e1,e2)     = reg(e1 .*) ∩ reg(e2  .*)
```

If `overlap(e1,e2)` does not match anything then we can freely flip between `e1|e2` and `e2|e1`. Also note that if this occurs then choice is deterministic and we do not have to backtrack if first alternative happens.

Mizushima [25] also transforms grammar to more deterministic one. We use stronger analysis. Using `overlap` we can determine where we can insert return states that inform `Switch` that next alternatives can not occur[2].

While bounds `minsize(e)`, `maxsize(e)` on minimal and maximal sizes of string that matches `e` can be discovered by intersecting with suitable languages it is faster to compute them by dataflow analysis.

Functor *reg* can be defined in the following way:

```
reg( 'c'                    ) = c
reg( r                      ) = reg(r)
reg( a* )                     = reg(a)*
reg( nested(start,mid,end)) = reg(start) .* reg(end)
reg( a   b )                  = reg(a)   reg(b)
reg( a | b )                  = reg(a) | reg(b)
reg(&a   b )                  = reg(a)  ∩ reg(b)
```

We use rough approximation of middle of `nested`. In typical case inside nesting could be practically anything so trying to improve this approximation leads only to larger expressions without any new insights.

We shall remark that better result can be obtained by first using relativized regular machine and then converting to regular machine. This gives two advantages: First is that `Switch` describes also lookaheads and we can describe intersection by lookahead.
Second is that we can use facts:
If `A` is unambiguous then `A B ∩ A C = A ( B ∩ C)`.
If `A` is unambiguous then `A B | A C = A ( B | C)`.

As there only finitely many $(continuation, cuts, stops)$ triples size of our machine is finite.

[2]We can also consider continuations for better results

2.8 Problems of left recursion

Left recursion handling deserves topic of its own. Various approaches were suggested and various counterexamples found.

In PEG implementing left recursion correctly is an impossible task. Consider rule:

```
L = &( L 'cd'   ) 'abc' # a -> abc -> abcbc -> ab
  | &( L 'bcd'  ) 'ab'  #       ^               |
  | L 'bc'              #       |               V
  | L 'cb'              #     abcbcb     <-   abcb
  | 'a'
```

On *"abcbcbcd"*.

It creates infinite cycle in the recursion. This problem is more fundamental as there is a paradox:

```
L = ~L
```

We reject such self references and raise an error when lookahead refers to possibly indirectly left recursive rule. Note that in boolean grammars same problem was recognized [29].

Left recursion can be handled by recursive descend/ascend. A rule:
`L = L 'bc' | L 'c' | 'ab' | 'a'`
on *"abc"* is recognized as *"(a(bc))"* by recursive descend parser but as *"((ab)c)"* by recursive ascend one. All previous approaches in PEG and context-free bottom-up parser used a recursive ascend variant of left recursion. A simplest algorithm is attributed to Paull [1]. It consist of rewriting direct left recursion to equivalent rule:

```
L = L a | b | L c | d
L =      (b |       d) (a|c)*.
```

An indirect left recursion is removed by inlining and thus reducing to direct recursion case.

In 1965 Kuno [21] suggested to limit recursion depth by n. It was rejected in PEG setting as in presence of semantic predicates some recursive rules need more than n calls. Also it was not clear how handle infinite streams. But it was rejected prematurely.

Using *reg* functor (or simple dataflow) we can for each expression compute lower bound on minimal length of a string that matches that expression. Using this information we can easily estimate minimal size of current continuation. When this bound exceeds the length of our string we can fail.

For infinite streams we can guess bound by guessing initially 1 and doubling bound when recursion could continue. We do not use this approach as it has an exponential complexity in the worst case.

Note that same technique can improve to Frost's algorithm [13].

In packrat setting Ford used Paull algorithm to remove direct left recursion. He rejected to support left recursion with the following reason [12]:

"At least until left recursion in TDPL is studied further, utilizing such a feature would amount to opening a syntactic Pandora's Box, which clearly defeats the pragmatic purpose for which the simple left recursion transformation is provided."

Warth, Douglass, Millstein [42] attempted to add runtime detection of left recursion. With bit of imagination it could be interpreted as doing Paull algorithm at runtime. However this approach has several flaws.

One discovered by Tratt [40] is that seed growing introduces ambiguity of direct left recursion when right recursive alternative is also present.

A revised algorithm of Tratt still contains a flaw. Tratt at certain times forbids expansion of right recursion.

Tratt approach fails to handle right-recursive lookahead as the following counterexample shows.

```
L = L 'a'
  | ~('b' L) 'b'
  | 'c'
```

Third issue was discovered by Peter Goodman [14]. Warth algorithm does not handle the following grammar.

```
A = A 'a'  / B
B = B 'b'  / A  / C
C = C 'c'  / B  / 'd'
```

Medeiros in unpublished paper [24] devised a revised version of seed growing algorithm.

One of possible advantages of seed growing could be support of higher order parametrized rules. In amethyst parser most of higher order rules are inlined making this point a moot one.

2.8.1 Left recursion in $\mathrm{REG}^{\mathrm{REG}}$ parser

We combine two techniques. First we just rewrite recursion by Paull algorithm. A second technique is that continuation passing style does implicit finite state machine minimization. This is simpler and leads to smaller grammars than Moore's left corner transform heuristic [26].

We handle left recursion inside iteration by unrolling one level.

With some bookkeeping we can transform left recursion to recursive descend. Idea is that each alternative returns its derivation and we choose a lexicographically smallest in recursion tree. This can be done in $O(1)$ time using dynamic lowest common ancestor [10].

2.9 State handling

We show how to implement techniques for state handling that we described in section 1.14.

We use a simple memoizing top-down PEG parser implemented in Ruby as an example. Implementation in amethyst uses similar ideas but intermixed with handling of other features.

```
class Match
  ["src","pos","result","contextual_arguments",
   "contextual_returns","locals"].each{|name|
    eval "
      def #{name} ( );@hash[\"#{name}\"]  ; end
      def #{name}=(v);@hash[\"#{name}\"]=v; end
    "
  }
  def timestamp ; deep_clone(@hash); end
  def revert(ts); @hash=ts         ; end
  def memo_id; [src,pos,deep_clone(contextual_arguments)];  end
  attr_accessor :memoized #we don't revert memo table
```

One could instead of deep cloning track what changes we did and revert them. This has same time complexity as without persistence because every revert will be payed by corresponding addition.

```
def match(exp)
  case exp
  when Call     id=memo_id + exp.name
                if !memoized[id]
                  ts =timestamp
                  locals, contextual_returns = {} , {}
                  r=match($rules[exp.name])
                  memoized[id]=deep_freeze(clone)
                  revert(ts)
                end
                c=memoized[id]
                result,pos=c.result,c.pos
                c.contextual_returns.each{|k,v|
                  contextual_returns[k] << v
                }
  when Char;    if src[pos]==exp.char   ;  result=exp.char; src+=1
                else                    ;  result=:fail
                end
  when Seq      exp.each{|seq|
                  if match(seq) == :fail;  return :fail
                  end
                }
  when Or       exp.each{|alt|
                  ts = timestamp
                  if match(alt) != :fail;  return result
                  else                   ;  revert(ts)
                  end
                }
                return :fail
  when Act      result=exp.call(self)
  when Bind     locals[exp.name]=result
  end
  return result
end
```

3. Dynamic parsing

Normal parser processes files in batch fashion. Amethyst allows dynamic parsing where the user is allowed to add and delete characters from string and query current parser output.

Editors and IDE try to maintain syntax highlighting and error detection often in ad-hoc way. Syntax highlighting typically uses regular expressions to determine meaning of text edited. This yields several problems, one is that regular expression can be confused with certain inputs. Other is updating regular expression for new versions of grammar. Dynamic parsing solves these problems in a robust way

In this chapter we develop a generic way how transform memoizing top-down parser to dynamic one. The update operation of the dynamic parser has the worst case time complexity $O(r \ln n)$ where r is number of rules that need recomputed. For typical workloads running time is $O(r)$. Our techniques can be applied to packrat parsers, parsing algorithm of Frost, and REG$^{\mathrm{REG}}$ parser.

Main idea of our approach is to annotate memoized rule with an interval of input used to calculate it. We use a balanced tree to detect if this interval changed or not. After receiving an update to the input we update version of corresponding element. When we need to recalculate memoized rule we check if there was a change in its interval and recalculate it as necessary.

3.1 Interface to memoizing top-down parsers

All three algorithms (REG$^{\mathrm{REG}}$, PEG, Frost's algorithm, ...) allows separation of memoization into independent module. In dynamic parsing this module serve as intermediary interface between user (IDE, editor, ...) and parser.

Our parser can be easily generalized to matching arrays of arbitrary type with no modications in algorithm.

A user interface consists of following four methods:

`chr(p)`	Character at position p
`ins(p,c)`	Insert character c at position p
`del(p)`	Delete character at position p
`parse`	Return result of parser

Interface with parser is more interesting. A parser can access string only by pointers that always point to same character regardless of modifications.

`char(ptr)`	Value of current character.
`next(ptr)`	Next character.
`get_memo(rule,ptr)`	Returns memoized value.
`set_memo(rule,ptr,value)`	Memoizies given value.

We assume that parser calls `get_memo` calls on entering rule and `set_memo` on exiting rule. Parser does not have to call `set_memo` if it decides not to memoize a rule.

3.2 Data structure

We present data structures for $O(r \ln n)$ time bound.

Our structure is a balanced tree. We maintain several properties:

`value`	value of character
`sons`	number of sons in subtree
`ts`	timestamp
`maxts`	maximal timestamp of son nodes
`memoized`	Memoization entries starting at this position.

We use timestamp that increases after call of each `parse`. Each insertion/deletion gets assigned this timestamp

Our implementation needs several auxiliary methods:

`timestamp(first,last)`	maximal timestamp in interval specified by first and last.
`index(ptr)`	position of character in current string.
`rindex(n)`	pointer to n-th character in current string.

Writing balanced tree that supports `chr,ins,del,timestamp,index,rindex` methods in $O(\ln n)$ while maintaining properties above is typical homework exercise. Note that queries that we made exhibit spatial locality. Thus a splay tree [37] looks like good candidate to obtain $O(1)$ running time in practice.

Implementing our data structure as tree with node for each character is unpractical. Observe that order in which we modify string between calls to parse method is not important. We can modify our data structure to rope data structure [6] with property that leaf substrings have same timestamp. When splitting substring we need also update table entries. As character can participate in at most $O(\ln n)$ splits amortized $O(\ln n)$ time complexity still holds.

There is a technical problem with the deletion. We need to save somewhere that deletion occurred. We keep nodes that contain no character for this purpose. Luckily we can always merge two empty adjacent nodes into one. It is easy to see that number of empty nodes will be at most the number of nonempty ones.

3.3 Algorithm

We will present two implementations. First implementation is an extension of amethyst that overwrites '.' operator. Second is in pseudocode that serves as overview of the effective `C` implementation from `lib/dynamic` subdirectory of the amethyst project.

```
amethyst Dynamic {
  init  = { @@memoized={}; @@rightmost=0}
  if(x) = &{x}

  memo(rule) =
      {id(rule,src,pos)}:id
      {pos}:oldpos
      {@@rightmost}:oldright
      {pos}:@@rightmost
      ( if(!memoized[id])
          (apply(rule) | {"failed"}):result
          { memoized[id]=Memo[pos-oldpos,@@rightmost-oldpos,result] }
        | -> nil
      )
      { pos=oldpos+memoized[id].advance
        @@rightmost=max(oldright,pos+memoized[id].rm_advance)
      }
      ( if(memoized[id].result=="failed")   fail
        |                                   -> memoized[id].result
      )

  anything = if(pos>=len) fail
           | {@@rightmost=max(@@rightmost,pos);pos=pos+1} -> src[pos-1]
  #seq is analogous
}
class Memo
  attr_accessor :advance,:radvance,:result
  def self.[](advance,radvance,result)
    m=Memo.new
    m.advance,m.radvance,m.result=advance,radvance,result
    m
  end
end
```

Our algorithm maintains stack that mirrors a call stack of parser. For each rule we find a rightmost position that can affect result of rule.

```
stack_struct stack
stack_push(rule,ptr){
  stack.push
  stack.top.rule=rule
  stack.top.ptr =ptr
  stack.top.last=ptr
}
stack_pop(rule,ptr){
  last=stack.top.last
  stack.pop
  update_last(last)
}
```

```
update_last(ptr){
  if (index(ptr)>index(stack.top.last))
    stack.top.last=ptr
}
```

There is technical issue that saving rightmost position as pointer is unwieldy. Instead we represent rightmost position as number of characters from starting position. With this improvement we for example do not have to worry what happens if rightmost position is deleted.

```
char(ptr){
  update_last(ptr)
  return ptr.value;
}
get_memo(rule,ptr){
  if (ptr.memoized[rule])
    last=rindex(ptr.index+ptr.memoized[rule].advance)
    if(timestamp(ptr,last)==ptr.memoized[rule].saved){
      update_last(last)
      return ptr.memoized[rule].value
    }
  }
  stack_push(rule,ptr)
  return nil
}
set_memo(rule,ptr,value){
  while (stack.top.rule!=rule ||
         stack.top.ptr!=ptr)
    stack_pop //parser decided not to memoize
  ptr.memoized[rule].value   = value
  ptr.memoized[rule].advance = stack.top.last.index
                             - ptr.index
  ptr.memoized[rule].saved   = timestamp(ptr,stack.top.last)
  stack_pop
}
```

4. Peridot

We use peridot as an example of using amethyst in language design.

4.1 Basic concepts

Peridot does not differ much from mainstream dynamic programming languages. We assume that reader is familiar with concepts like class, method, dynamic dispatching.

Currently Peridot has classes for integers, arrays and strings with basic methods and operators. For their description see Peridot documentation.

As in Ruby variables are defined by assignment.

4.2 Peridot grammar in amethyst

We use a parts of Peridot grammar to illustrate use of amethyst. Entire grammar can be found in Appendix C.

We try to design an operator precedence that avoids pitfalls of `C` language that expressions `1 + 1<<2 == 5` and `1&2 == 5&2` are false.

```
binary_op(exp,oper) = apply(exp):a ("" apply(oper):op
                      apply(exp):b {call(op,a,b)}:a)* -> a

expr_or_l  = binary_op('expr_and_l',(| "||" |))
expr_and_l = binary_op('expr_cmp'  ,(| "&&" |))

expr_cmp = binary_op('expr_ar1',(| '<' |'<='|'<=>'
                           |'>='| '>' |'=='|'!=' |))

expr_ar1 = binary_op('expr_ar2',(| '+' |'-'        |))
expr_ar2 = binary_op('expr_or' ,(| '*' |'/' |'%'   |))

expr_or  = binary_op('expr_and',(| '|' |'^'        |))
expr_and = binary_op('expr_ar3',(| '&'             |))

expr_ar3 = binary_op('expr_prefixed',(| '**'|'<<'|'>>' |))

prefix_op(exp,oper) = apply(o):op apply(exp):e -> call(op,e)

expr_prefixed = "+" expr_prefixed
              | prefix_op('expr_prefixed',(| '-' |))
              | expr_postfixed
```

4.3 Functional programming style

Peridot supports several functional programming features.

Lambdas are supported with same syntax as in amethyst.

Ruby extensively uses the block passing style. You use it even for loops:

```
4.times{|i|
  puts i
}
#equivalent code
4.times(&proc{|i| puts i})
```

This construction can be viewed as a case of the continuation passing style [39].

We want have better support for continuation passing style.

In Peridot a yield keyword returns a (result, continuation) pair. The block syntax `a(b){block}` is a shortcut for:

```
cont = a(b)
  while (cont.is_a?(Continuatation))
    r    = block.call(cont.result)
    cont = cont.call(r)
  end
```

When no block is specified you can use returned `Continuation` object is similar way as `Enumerator` object in Ruby. A main difference is that in `Continuation` object communication goes in both directions.

As hypothetical example consider a binary search tree that implements a generic binary search `bsearch` method that takes a block with supplied comparison method. For example guess a number game can be done as:

```
t=Tree.new
t.bsearch{|x|
  puts "is"+x+"more/less/equal to your number?"
  gets
}
```

We can change numbers passed to block by map method:

```
t=Tree.new
c=t.bsearch
c=c.map{|x| roman_numeral(x) }
c.call{|x|
  puts "is"+x+"more/less/equal to your number?"
  gets
}
```

And values returned back. For example reversing direction can be done by changing third line of previous example to:

```
c=c.rev_map{|x|
  if x=="more"
    "less"
  else
    if x=="less"
      "more"
    else
      "equal"
    end
  end
}
```

A. Amethyst syntax summary

In this section we recapitulate semantic of amethyst in systematic manner.

Rules

Pattern	Description
`rule = exp`	Rule definition.
`rule`	Rule call.
`rule(v1,v2...) = exp`	Parametrized definition.
`rule(c1,c2...)`	Parametrized call.
`Grammar::rule`	Call rule from given Grammar.
`(| e |)`	Lambda

Semantic actions an predicates

Pattern	Shortcut	Description
`{c}`	core	Semantic action.
`&{c}`	`core`	Semantic predicate.
`~{c}`	`&{!c}`	Negative semantic predicate.
`-> c newline`	`{c}`	Alternative syntax of semantic action.

In semantic actions and predicates we do following substitutions.

Pattern	Shortcut	Description
`@method`	`src.method`	method call (parameters are allowed)
`@Class`	`Class.create(...)`	Construct object
`@>name`	`arguments["name"]`	Contextual argument
`@<name`	`returns["name"]`	Contextual return
`(|e|)`	core	Lambda

Variable binding

Pattern	Shortcut	Description
`e:v`	core	variable binding
`e:{c}`	`e:it {c}`	For conversions etc.
`e:[c]`	`e:{v << it}`	Append result to array `c`.

Sequencing and choice

Operation	Expansion	Description
`e?`	`e|{nil}`	Make `e` optional.
`Cut`	auxiliary	Like ! in prolog
`~e`	`e Cut fails | {nil}`	Negative lookahead.
`e1 & e2`	`~~e1 e2`	Positive lookahead.

Iteration

Pattern	Shortcut	Description
`e**`	auxiliary	repeat-until
`Stop`	auxiliary	Stop iteration
`e*`	`e**`	When `e` contains `Stop`,
`e*`	`(e|Stop)**`	otherwise.
`break`	`Cut Stop`	Possible expansion.

Object orientated constructs

Pattern	Shortcut	Description
`e1[e2]`	core	Enter operator
`[e]`	`.[e]`	We can omit leading . .
`e1=>e2`	`e1:a {[a]}[e2]`	Pass operator
`Class`	`member(Class)`	Test class membership

B. Standard prologue

```
amethyst Amethyst < AmethystCore {
  space = <\s\t\r\n\f>
  spaces= space*
  token(s) = spaces seq(s)
  _      = space

  lower  = <a-z>
  upper  = <A-Z>
  alpha  = lower | upper
  alnum  = alpha | digit
  digit  = <0-9>
  xdigit = <0-9a-fA-F>
  word   = alpha | '_'

  newline =  '\r\n' | '\r' | '\n'
  line = (newline break | .)*:{it*""}

  empty = -> nil
  eof= ~.

  seq(s) = _seq(s) {s}

  int = ('-'|{""}):s ( '0x' <0-9a-fA-F>+ | '0b' <01>+
                     | '0o' <0-7>+ | <0-9>+)[]:n {(s+n*"").to_i}
  number = int


  find(exp)    = (apply(exp):e break | .)* .* -> e
  replace(exp) = (apply(exp)         | .)*:{it*""}

  until(e)     = ( seq(e) break
               | ('\\':[x])? .:[x]
               )* -> x.join

  listOf(rule,del) = apply(rule):[f] (seq(del) apply(rule))*:[*f] ->f
                   | empty -> []

  reverse(l) = {@@rev||=Hash.new{|h,k| h[k]=k.reverse }}
               _reverse(@@rev[@self])
               apply(l):rev
               _reverse(@@rev[@self])
              {rev}

  fails = &{false}

  char= .:c &{c.is_a? String } -> c

  member(x)     = .:a &{x === a} {a}
  true          = member(true)
  false         = member(false)
  nil           = member(nil)
  clas(cls)     = member(cls)
  range_in(a,b) = member(a.. b)
  range_ex(a,b) = member(a...b)
  regch(regex)  = member(regex)

  parse(rule,obj,a) = {obj}[{apply(rule,*a)}:r {self.prof_report;r}]

  nested(start,mid,end) = seq(start) apply(mid) seq(end)
}
```

C. Peridot grammar

```
class Peridot_parser < Amethyst
  def call(name,*args)
    Call[{:name=>leterize(name),:ary=>args}]
  end
end
amethyst Peridot_parser{
  root = (body | defi | sequence)*:a .* {a}

  body = "class" "" name:name defi*:ary "end" @Klass

  defarg= <^,)>+:{it.join}

  defi = "def" "" defname:name {["obj self"]}:args
         '(' listOf('defarg',','):[args] ')'
         sequence:[ary] "end" @Def
  name = <a-zA-Z_>:s <a-zA-Z0-9_>*:{s+it*""}
  defname = (<^ \t\r\n()>)*:x {leterize(x*"")}

  atom = ""
       ( number:n          -> CCode["Int(#{n})"]
       | '"' until('"'):s -> CCode["Str(#{s.inspect})"]
       | '(|' expr:e "|)" -> Lambda[e]
       | '(' expr:e ")" {e}
       | 'if' "(" expr:expr ")" block:block
             -> If[{:expr=>expr,:block=>block}]
       | '{' ('}' break | &'{' atom:{'{'+it[0]+'}'}:[s] | .:[s])*
             -> CCode[s*""]
       | 'yield' atom:a   -> Yield[a]
       | method("self")
       | local
       )

  local =  ~'end' name:name @Var

  args = listOf('expr',',')
  method(obj) = name:name '(' (args | {[]}):arg ")"
                    ( block:b  -> Iterate[call(name,obj,*arg),b]
                    |          -> call(name,obj,*arg)
                    )

  expr_postfixed = atom:a (
       ( '[' args:arg "]" "=" expr:arg2 -> call("[]=",a,*arg,arg2)
       | '[' args:arg "]"               -> call("[]",a,*arg)
       | '.' method(a)
       | '.' name:name                  -> call(name,a)
       ):a
      )* {a}

  expr = expr_ass


  binary_op(exp,oper) = apply(exp):a
           ("" apply(oper):op apply(exp):b {call(op,a,b)}:a)* -> a

  expr_ass = "" name:name '=' expr:expr @Assign
           | expr_or_l
```

```
expr_or_l  = binary_op('expr_and_l',(| "||" |))
  expr_and_l = binary_op('expr_cmp'  ,(| "&&" |))

  expr_cmp = binary_op('expr_ar1',(| "<" |"<="|"<=>"|
                                   ">="|">" |"=="|"!=" |))
  expr_ar1 = expr_ar2:a (("+" expr_ar2
                          |"-" expr_ar2:{call('-',it)}):b
                         {call('+',a,b)}:a   )* -> a
  expr_ar2 = binary_op('expr_or' ,(| '*' |'/' |'%'  |))
  expr_or  = binary_op('expr_and',(| '|' |'^'       |))
  expr_and = binary_op('expr_ar3',(| '&'            |))
  expr_ar3 = binary_op('expr_pfx',(| '**'|'<<'|'>>' |))

  prefix_op(oper,exp) = apply(oper):op apply(exp):e -> call(op,e)

  expr_pfx =             "+"    expr_pfx
           | prefix_op((|'-' |),'expr_pfx')
           | prefix_op((|<!~>|),'expr_pfx')
           | expr_postfixed

  block = "{" sequence:s '}' {s}

  sequence = expr:[ary] ( newline expr:[ary])* @Seq
}
```

Bibliography

[1] Aho V., Sethi R., Ullman J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts. 1986.

[2] Backus J. W., Bauer F. L., Green J., Katz, C., McCarthy J., Perlis A. J., Rutishauser H., Samelson K. et al. , Naur Peter ed. *Report on the Algorithmic Language ALGOL 60.* Copenhagen. , May 1960

[3] `http://kam.mff.cuni.cz/~ondra/amethyst/parser_highlight.ame.html`.

[4] `http://kam.mff.cuni.cz/~ondra/benchmark_string`.

[5] `http://kam.mff.cuni.cz/~ondra/regreg`.

[6] Boehm H-J., Atkinson R., Plass M. *Ropes: an Alternative to Strings*, Software—Practice Experience (New York, NY, USA: John Wiley Sons, Inc.) 25 (12): 1315–1330 and computation, Addison-Wesley, Reading (1979).

[7] Burge W. H. *Recursive Programming Techniques,* The Systems programming series. Addison-Wesley. 1975

[8] Chomsky N. *Three models for the description of language*, Information Theory, IEEE Transactions 2 (3): 113–124, September 1956

[9] Church A *A set of postulates for the foundation of logic*, Annals of Mathematics, Series 2, 33:346–366 (1932)

[10] Cole R., Ramesh H. *Dynamic LCA queries on trees,* Proceeding SODA '99 Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms

[11] Driscoll J. R., Sarnak N., Sleator D. D., Tarjan R. E., *Making Data Structures Persistent*, Journal of Computer and System Sciences, Vol. 38, No. 1, 1989

[12] Ford Bryan, *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*, Master's thesis, MIT, September 2002

[13] Frost R., Hafiz R., Callaghan P. *Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars,* 10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE (Prague), June 2007

[14] Goodman P, *Re: [PEG] Res: Problem w/ nullable left recursion and trailing rules in "Packrat Parsers Can Support Left Recursion"* `http://www.mail-archive.com/peg@lists.csail.mit.edu/msg00185.html`, 2008

[15] Greif I. *Semantics of Communicating Parallel Processes. Ph.D. thesis,* Technical Report MAC-TR-154, Project MAC, MIT (Cambridge), September 1975

[16] Grimm R. *Better extensibility through modular syntax,* In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI '06), pp. 38–51, June 2006

[17] Hopcroft J. E., Ullman J. D. *Introduction to automata theory*, languages

[18] Johnson M. *Memoization in top-down parsing,* Comput. Linguist., 21(3): 405-417, 1995.

[19] Kleene S. C. *Representation of Events in Nerve Nets and Finite Automata*, In Shannon, Claude E.; McCarthy, John. Automata Studies. Princeton University Press. pp. 3–42, 1956

[20] Kildall G. *A Unified Approach to Global Program Optimization,* Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages: 194–206., 1973

[21] Kuno S, *The predictive analyzer and a path elimination technique,* Comm. ACM 8(7) 453–462, 1965.

[22] McCulloch W. S., Pitts E. *A logical calculus of the ideas imminent in nervous activity*, Bulletin of Mathematical Biophysics: 541–544, 1943

[23] Medeiros Se., Mascarenhas F., Ierusalimschy R., *From Regular Expressions to Parsing Expression Grammars,* SBLP, September 2011.

[24] Medeiros S. *Left Recursion in PEGs* `http://www.lua.inf.puc-rio.br/~sergio/leftpeglist.pdf`, 2010

[25] Mizushima K., Maeda A., Yamaguchi Y.: *Packrat parsers can handle practical grammars in mostly constant space*, Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010 (S. Lerner, A. Rountev, Eds.), ACM, 2010.

[26] Moore C. *Removing left recursion from context-free grammars*, In Proc. 1st North American chapter of the Association for Computational Linguistics conference, pages 249–255, 2000.

[27] Norvig P. *Techniques for Automatic Memoization with Applications to Context-Free Parsing,* Computational Linguistics, Vol. 17 No. 1, pp. 91–98, March 1991

[28] Okhotin A. *Boolean grammars* , Information and Computation, Volume 194, Issue 1, 10 October 2004

[29] Okhotin A. *LR parsing for Boolean grammars*, International Journal of Foundations of Computer Science 17:3 (2006), 629–664.

[30] Rabin M. O., Scott D. *Finite automata and their decision problems*, IBM J. Res. Dev. vol 3, 1959

[31] Reynolds J. C. *Definitional interpreters for higher order programming languages*, In ACM Conference Proceedings, 1972.

[32] Redziejowsky R. *BITES instead of FIRST for Parsing Expression Grammar*, Fundamenta Informaticae 109, 3 (2011) 323-337.

[33] Seaton G. S. *A Programming Language Where the Syntax and Semantics Are Mutable at Runtime*, University of Bristol, Master thesis, `http://www.chrisseaton.com/katahdin/katahdin.pdf`

[34] Ryder B. G. *Incremental data flow analysis*, In Conference Record 10th Annual ACM Symposium on Principles of Programming Languages (Austin, TX): 167-176, ACM, New York, Jan. 1983

[35] Tanter E. *Contextual values*, Proceeding DLS '08 Proceedings of the 2008 symposium on Dynamic languages

[36] Tomita M. *An efficient augmented-context-free parsing algorithm*, Comput. Linguist. 13, 31–46, 1987

[37] Sleator D. D., Tarjan R. E. *Self-Adjusting Binary Search Trees*, Journal of the ACM (Association for Computing Machinery) 32, 1985

[38] Søndergaard, Harald, Sestoft *Referential transparency, definiteness and unfoldability*, Acta Informatica 27 (6): 505–517, 1990

[39] Sussman G. J., Steele G. L. Jr. *Scheme: An Interpreter for Extended Lambda Calculus*, MIT AI Lab. AI Lab Memo AIM-349. December 1975.

[40] Tratt L., *Direct Left-recursive Parsing Expression Grammars*, Technical report EIS-10-01, Middlesex University, October 2010

[41] Valiant L. G. *General context-free recognition in less than cubic time*, Journal of Computer and System Sciences 10, 1975

[42] Warth A. *Experimenting with Programming Languages*, PhD dissertation, University of California, Los Angeles, 2009 Alessandro Warth and Ian Piumarta, *OMeta: an Object-Oriented Language for Pattern Matching*, Dynamic Language Symposium 2007, October 2007.