ruby-tree-sitter

A Comprehensive Introduction to Tree-sitter

Derek Stride

Senior Software Engineer
@ Shopify

Tree-sitter

Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited.

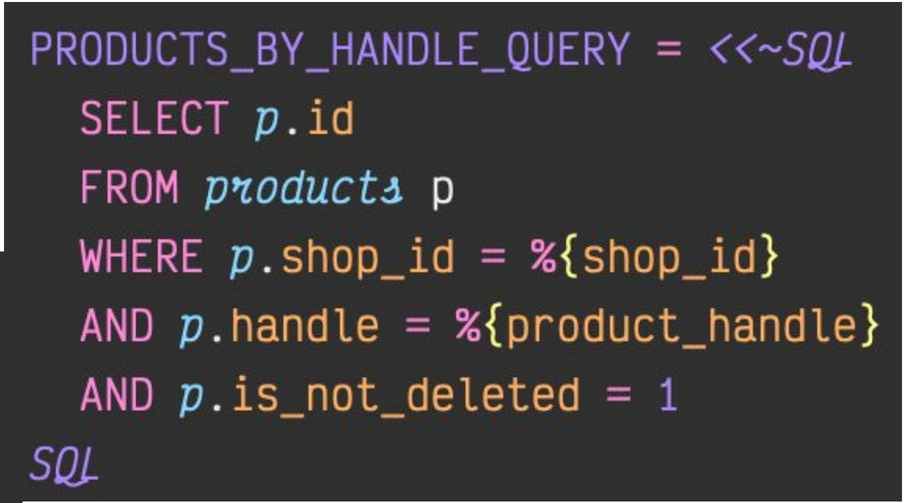
Tree-sitter

Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and **efficiently update the syntax tree** as the source file is **edited**.

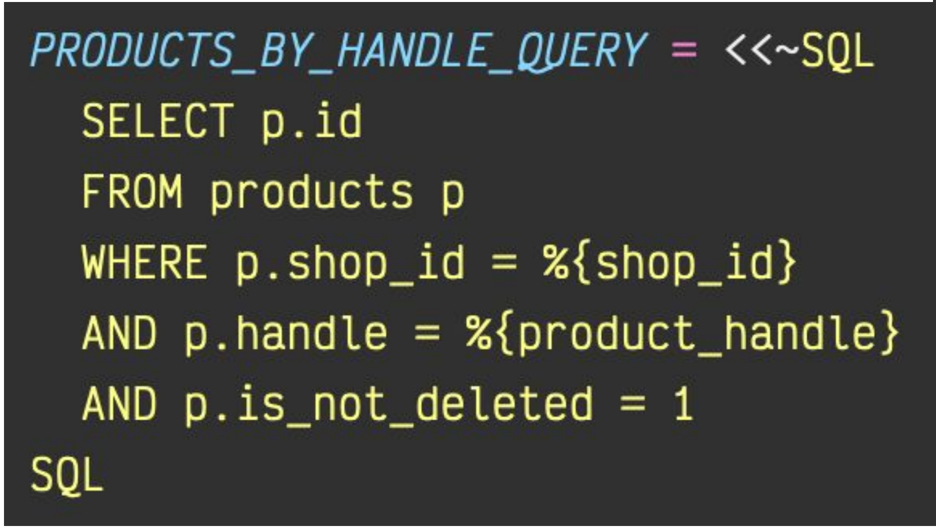
There are a ton of existing Grammars

`github://DerekStride/tree-sitter-sql`

Why Tree-sitter? Well, tree-sitter-sql.



```
PRODUCTS_BY_HANDLE_QUERY = <<~SQL
  SELECT p.id
  FROM products p
  WHERE p.shop_id = %{shop_id}
  AND p.handle = %{product_handle}
  AND p.is_not_deleted = 1
SQL
```



```
PRODUCTS_BY_HANDLE_QUERY = <<~SQL
  SELECT p.id
  FROM products p
  WHERE p.shop_id = %{shop_id}
  AND p.handle = %{product_handle}
  AND p.is_not_deleted = 1
SQL
```

Tree-sitter

Tree-sitter is a parser generator tool **and** an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited.

Generated vs Hand-written

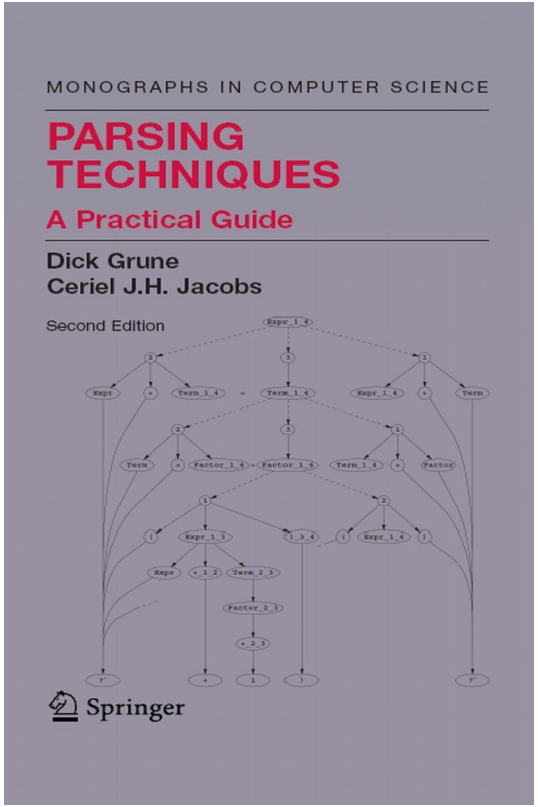
Hand-rolled versus generated

I will start this section by noting that all of the way back to version 0.76 in 1995 there was a ToDo file in the root of the repository that contained the line **hand written parser (recursive descent). I am personally very biased toward a hand-written recursive descent parser.** I believe this will allow us to have maximal control over error tolerance, I believe it will provide the most opportunity for documentation and testability, and I believe it will lead to the most maintainable parser going forward.

We can look to other languages as examples as well. Of the 2021 Redmonk top 10 languages, 8 of them use a handwritten parser. There are many pull requests and changesets linked in that blog post. I'll just call out the golang one that shows that by switching away from yacc-based and writing a hand-written recursive descent parser they reduced average parse time by 18%.

To be clear, I think we could accomplish all of the stated goals for this project and still use a generated parser. Plenty of tools use generated parsers to great effect (notably SQL parsers all tend to be generated), and we could leverage a lot of the work that Sorbet has done to achieve error tolerance. That being said, I still believe hand-writing it is the way to go.

Parsing Techniques - A Practical Guide



MONOGRAPHS IN COMPUTER SCIENCE

PARSING TECHNIQUES

A Practical Guide

Dick Grune
Ceriel J.H. Jacobs

Second Edition

Springer

11.3.2 Strong-LL(1) versus LALR(1)

For two linear-time methods, strong-LL(1) and LALR(1), parser generators are readily available, both as commercial products and in the public domain. Using one of them **will in almost all cases be more practical and efficient than writing your own**; for one thing, writing a parser generator may be (is!) interesting, but doing a reasonable job on the error recovery is a protracted affair, not to be taken on lightly. (Grune and Jacobs 251,252)

Llrama

Lrama is LALR (1) parser generator written by Ruby. The first goal of this project is providing error tolerant parser for CRuby with minimal changes on CRuby parse.y file.

Prism Ruby parser

This is a parser for the Ruby programming language. It is designed to be portable, error tolerant, and maintainable. It is written in C99 and has no dependencies.

Parsers

LR parsing / Bottom Up

- Shift-Reduce Parsers
- Uses a **state table** to determine which action to take.
- Uses grammar definitions to build state tables.

LL parsing / Top Down

- Most common Top down is recursive descent.
  - Recursive descent parsers are handwritten not generated.
- Uses the **call-stack** to maintain implicit state.

LR Parsers

- Shift-Reduce Parsers
- Uses a state table to determine which action to take.
- Uses **grammar definitions** to build **state tables**.

Example Grammar

Math Grammar – EBNF vs Tree-sitter (javascript)

```
expression
    : term
    | expression '+' term
    | expression '-' term
    ;
term
    : factor
    | term '*' factor
    | term '/' factor
    | term '%' factor
    ;
factor
    : primary
    | '-' factor
    | '+' factor
    ;
primary
    : IDENTIFIER
    | INTEGER
    | '(' expression ')'
    ;
```

```
_expression: $ => choice(
   $.variable,
   $.number,
   $.sum,
   $.subtraction,
   $.product,
   $.division,
   $.exponent,
   $._parenthesized_expression,
 ),

 sum: $ => prec.left(
   "addition",
   seq(
     field("left", $._expression),
     "+",
     field("right", $._expression),
   ),
 ),
```

Server-Timing

github://DerekStride/tree-sitter-server_timing

// A single metric
Server-Timing: <timing-metric>
// Multiple metrics as a comma-separated list
Server-Timing: <timing-metric>, …, <timing-metricN>

<timing-metric>

<name>

A name token (no spaces or special characters) for the metric that is implementation-specific or defined by the server, like cacheHit.

<duration> Optional

A duration as the string **dur**, followed by **=**, followed by a value, like **dur=23.2**.

<description> Optional

A description as the string **desc**, followed by **=**, followed by a value as a token or a quoted string, like **desc=prod** or **desc="DB lookup"**.

Server-Timing - Examples

- `cacheHit`

- `cacheHit;desc=”Powered by Redis”`

- `redis;dur=4.3;desc=”RoundTrips:3”, memcached;desc=”RoundTrips:2;dur=1.2`

- `db;desc=mysql, cacheMiss`

Server-Timing

```
// A single metric
Server-Timing: <timing-metric>
// Multiple metrics as a comma-separated list
Server-Timing: <timing-metric>, ..., <timing-metricN>
```

```
header: $ => seq(
  $.timing_metric,
  repeat(
    seq(",", $.timing_metric),
  ),
),
```

Server-Timing

<timing-metric>

<name>

A name token (no spaces or special characters)

<duration> Optional

A duration as the string **dur**, followed by **=**, followed by a value, like **dur=23.2**.

<description> Optional

A description as the string **desc**, followed by **=**, followed by a value as a token or a quoted string, like **desc=prod** or **desc="DB lookup"**.

```
token: _ => /[a-zA-Z]+/,


duration: $ => seq("dur=", $.number),
number: _ => /[0-9]+(\.[0-9]+)?/,


description: $ => seq("desc=", choice($.token, $.string)),
string: _ => choice(/"[^"]*"/, /'[^']*'/),
```

Server-Timing

<timing-metric>

<name>

A name token (no spaces or special characters)

<duration> Optional

A duration as the string **dur**, followed by **=**, followed by a value, like **dur=23.2**.

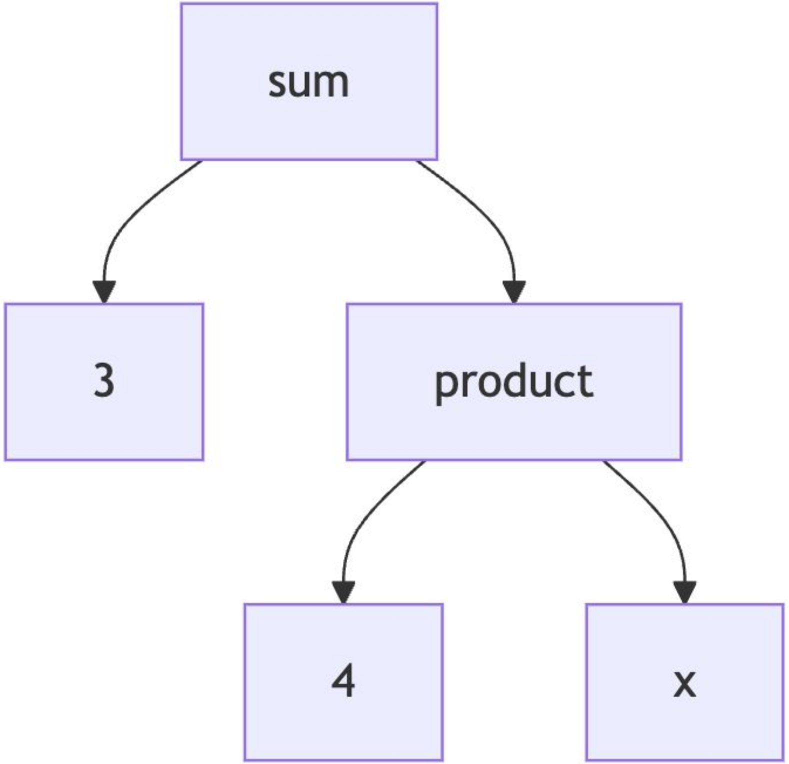
<description> Optional

A description as the string **desc**, followed by **=**, followed by a value as a token or a quoted string, like
desc=prod or **desc="DB lookup"**.

```
timing_metric: $ => seq(
  field("name", $.token),

  optional(
    choice(
      seq(";", $.duration),
      seq(";", $.description),
      seq(";", $.duration, ";", $.description),
      seq(";", $.description, ";", $.duration),
    ),
  ),
),
```

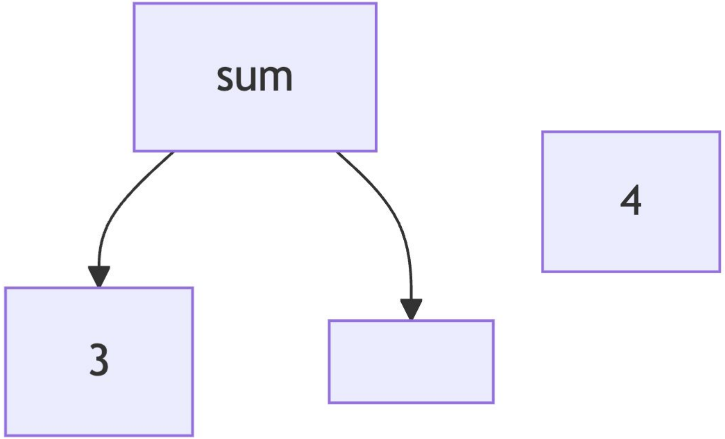
Building the Syntax Tree

Parsers – 3 + 4 * x



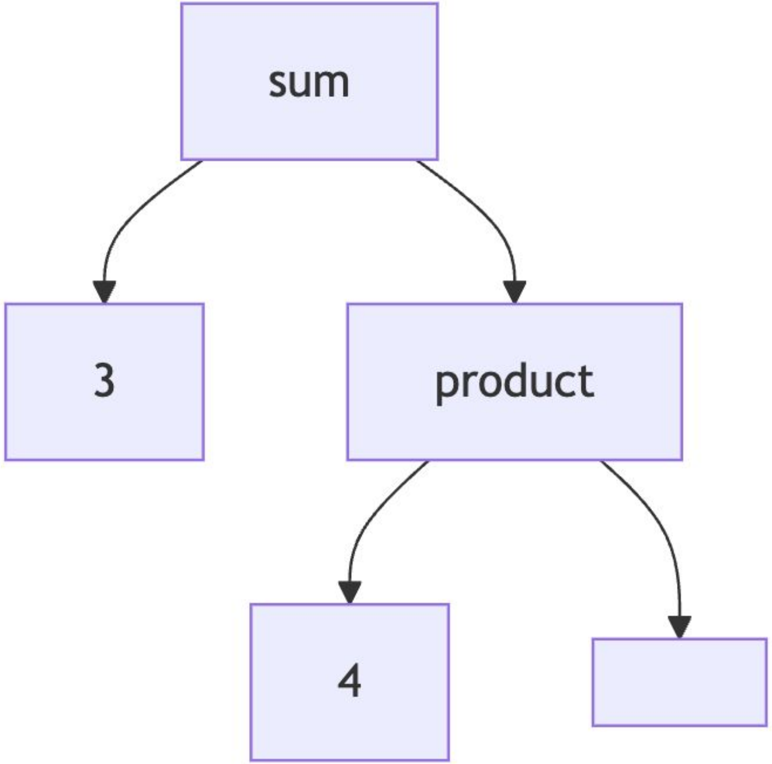
Parsers – 3 + 4 | * x

LL parsing / Top Down



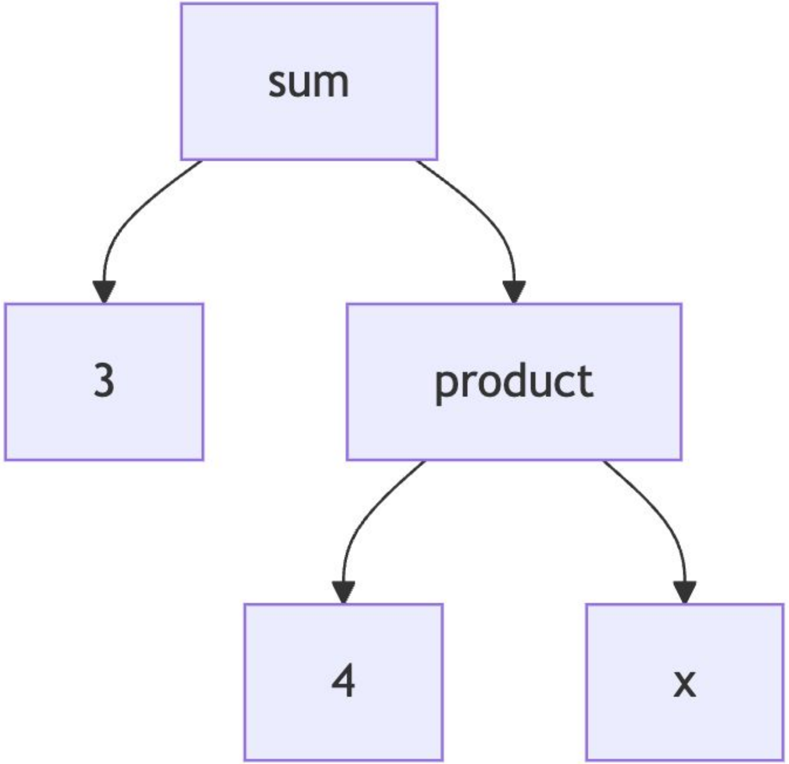
Parsers – 3 + 4 * | x

LL parsing / Top Down



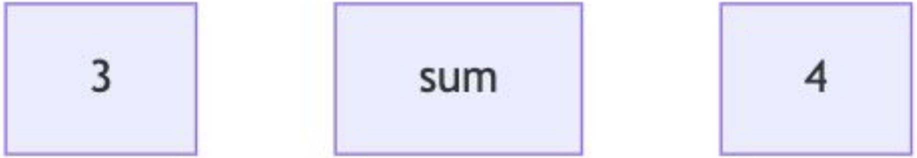
Parsers – 3 + 4 * x

LL parsing / Top Down



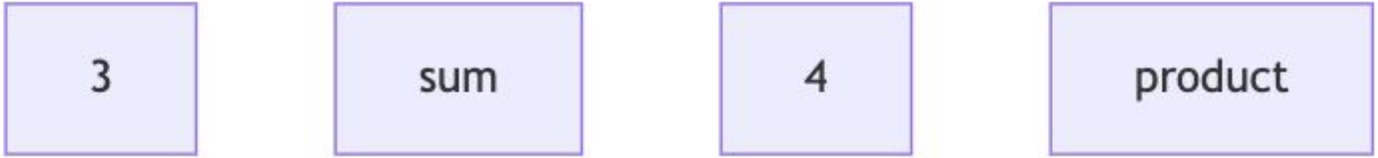
Parsers – 3 + 4 | * x

LR parsing / Bottom Up



Parsers – 3 + 4 *| x

LR parsing / Bottom Up



3

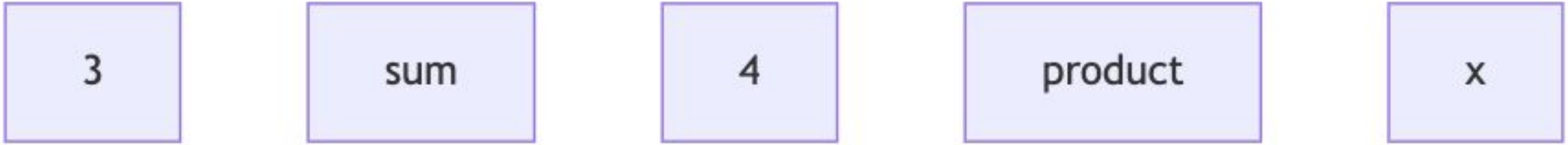
sum

4

product

Parsers – 3 + 4 * x

LR parsing / Bottom Up



3

sum

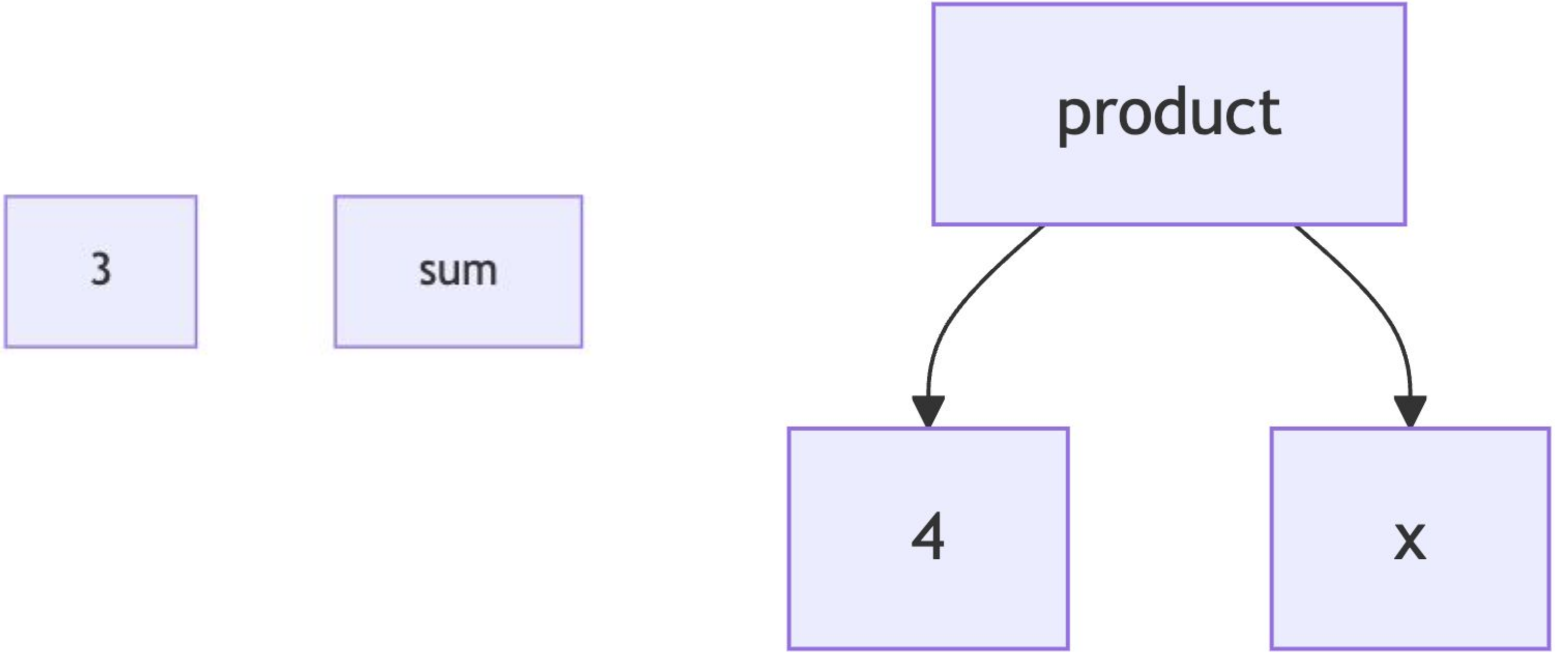
4

product

x

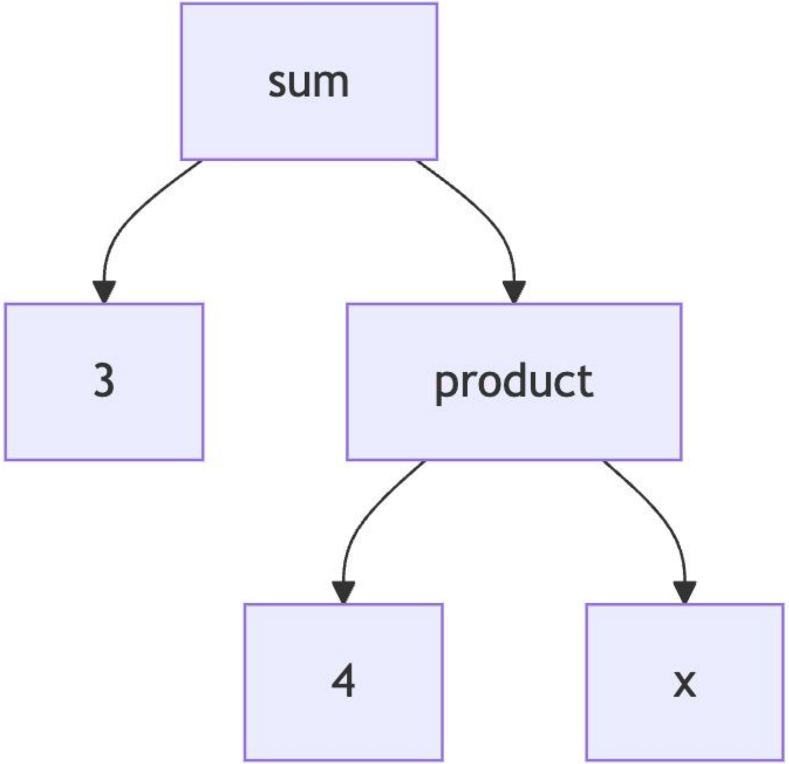
Parsers – 3 + 4| * x

LR parsing / Bottom Up



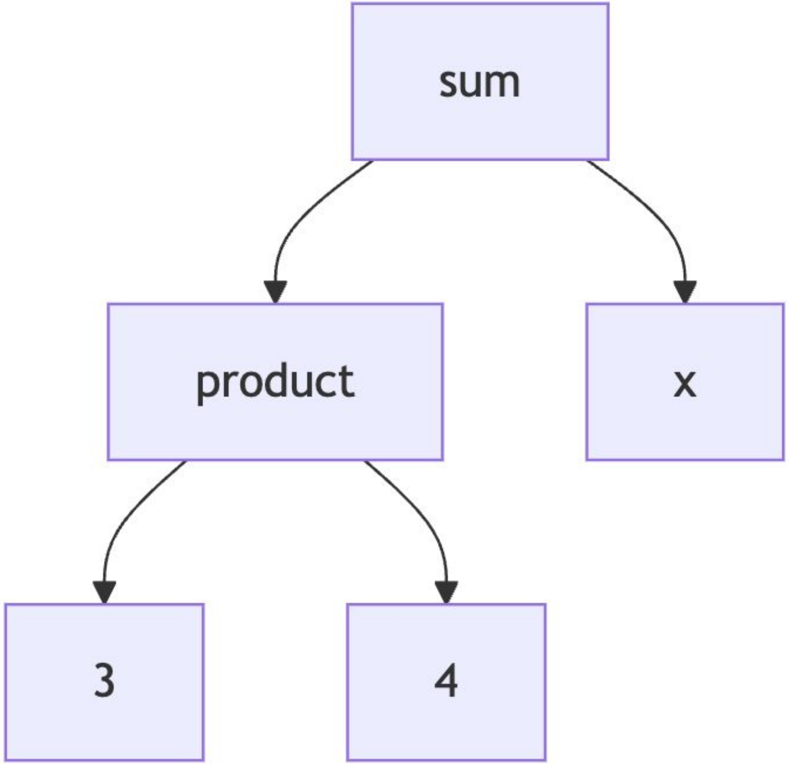
Parsers – `3 + 4 * x`

LR parsing / Bottom Up



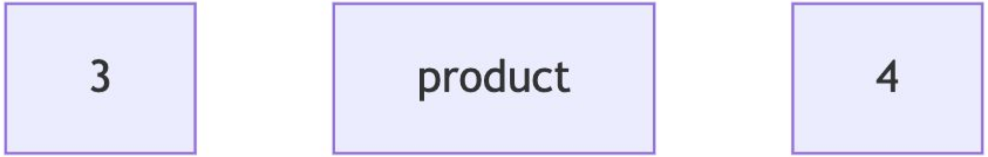
Parsers – 3 * 4 + x

LR parsing / Bottom Up



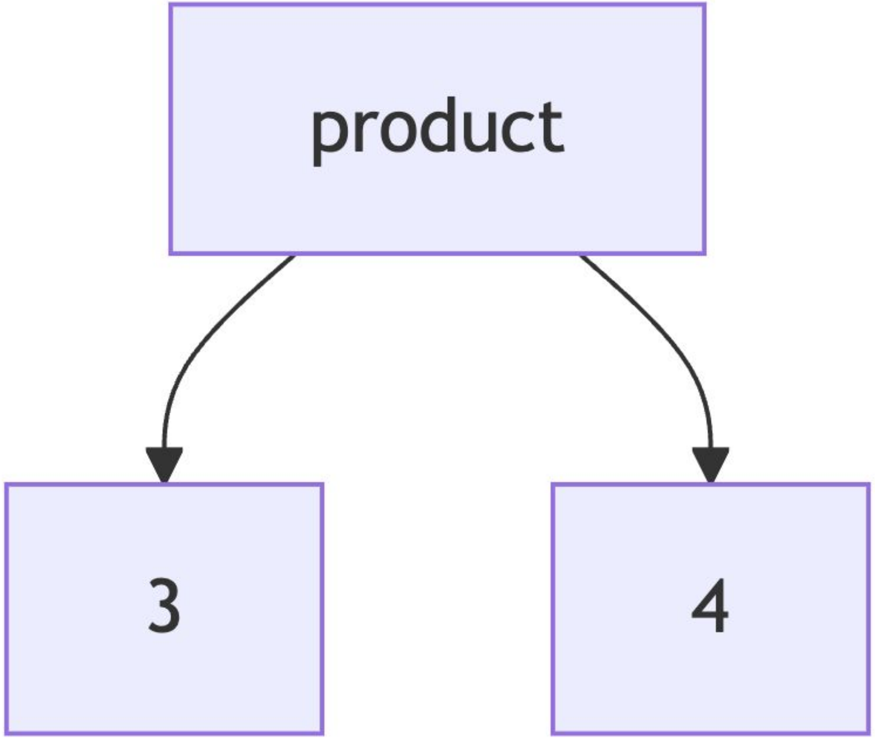
Parsers – 3 * 4| + x

LR parsing / Bottom Up



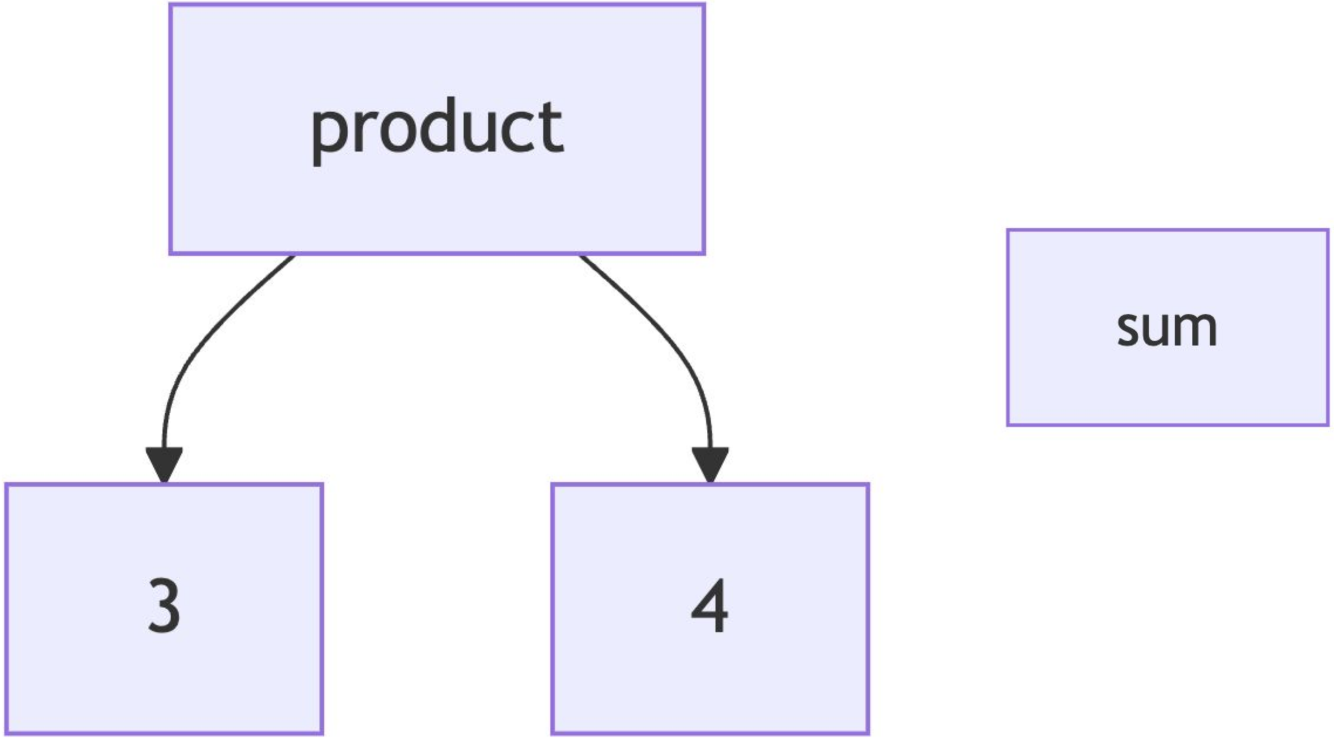
Parsers – 3 * 4| + x

LR parsing / Bottom Up



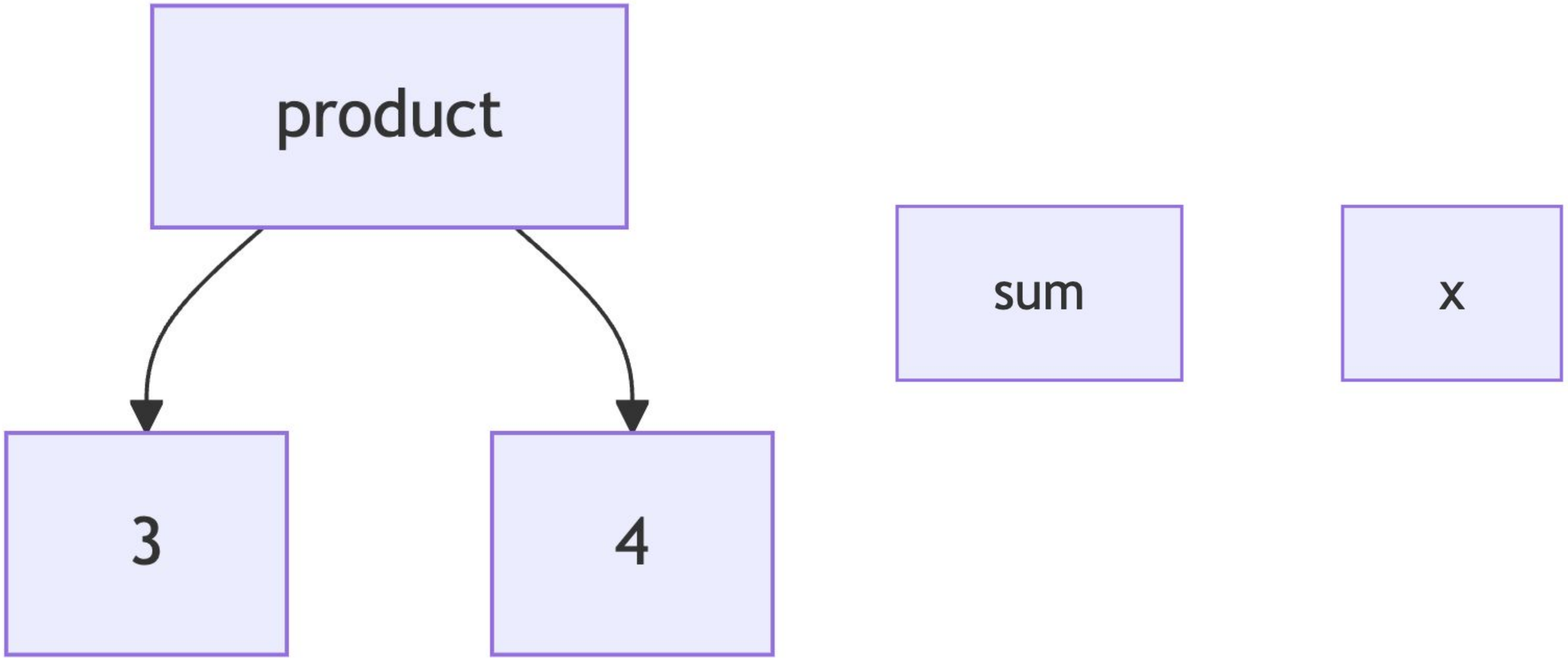
Parsers – 3 * 4 +| x

LR parsing / Bottom Up



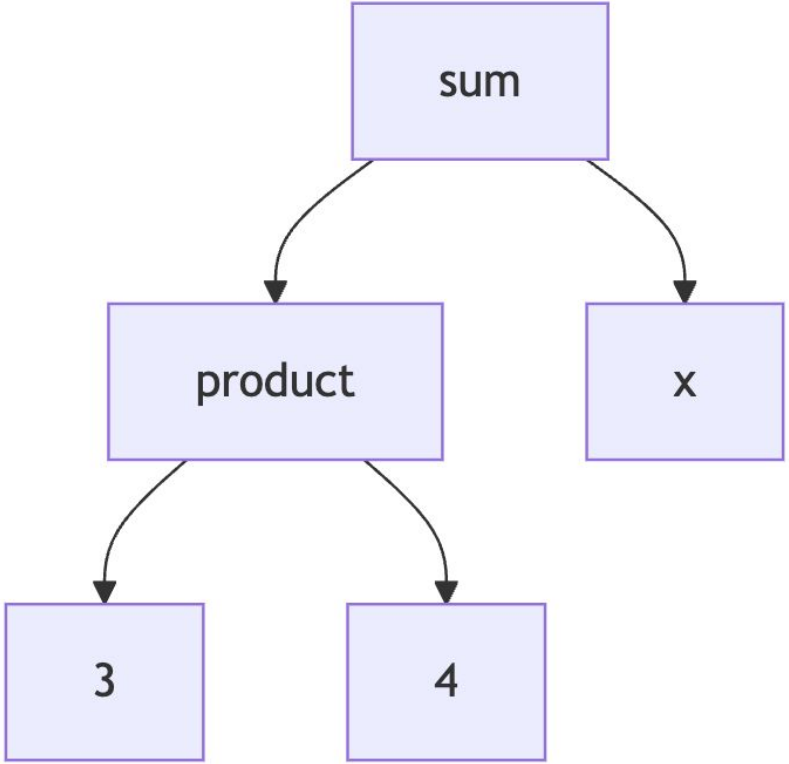
Parsers – 3 * 4 + x|

LR parsing / Bottom Up



Parsers – 3 * 4 + x

LR parsing / Bottom Up



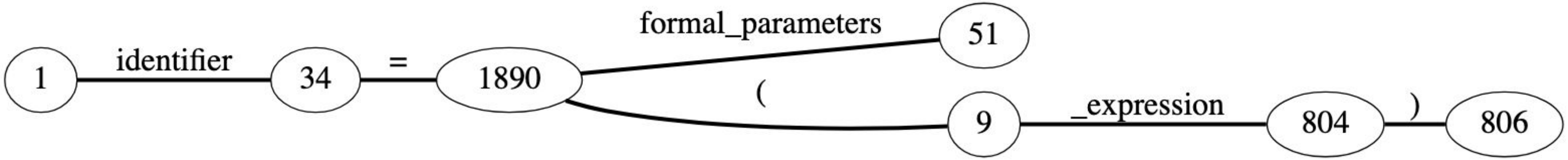
Aside: Ambiguities

Ambiguities

```
x = (y);       // parenthesized expression
     ^ expression


x = (y) => z;  // arrow function
     ^ parameter
```

Ambiguities - GLR

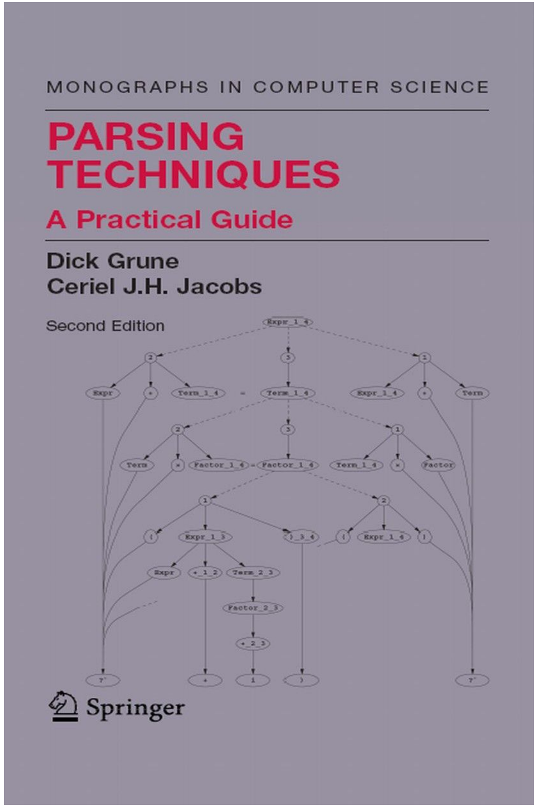
- Fork the parse tree and continue until alternative branch can be discarded.



Ambiguities - LR(*k*)

```
x = (y);       // parenthesized expression
       ^ Look-ahead k tokens
         Figure out next state to jump to
         Finally, backtrack

x = (y) => z;  // arrow function
        ^ Look-ahead k tokens
```

Parsing Techniques - A Practical Guide



MONOGRAPHS IN COMPUTER SCIENCE

PARSING TECHNIQUES

A Practical Guide

Dick Grune
Ceriel J.H. Jacobs

Second Edition

Springer

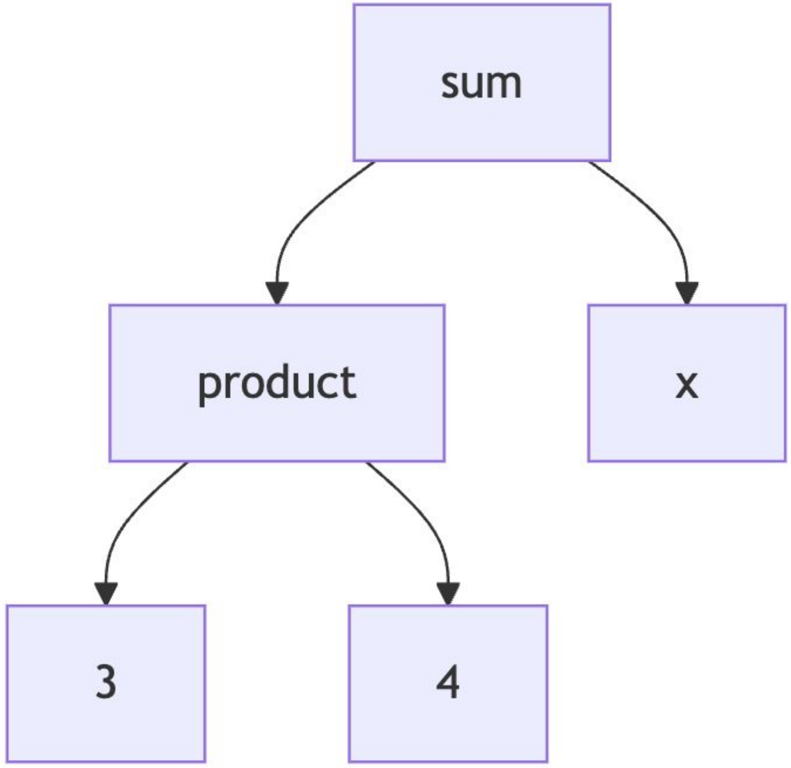
9.5.2 Some properties of LR(*k*) parsing

Instead of a look-ahead of one token, *k* tokens can be used. It is not difficult to do so but it is extremely tedious and the resulting tables assume gargantuan size (see, e.g., Ukkonen [LR 1985]). Moreover it does not really help much. Although an LR(2) parser is more powerful than an LR(1) parser, in that it can handle some grammars that the other cannot, the emphasis is on “some”. **If a common-or-garden variety grammar is not LR(1), chances are minimal that it is LR(2) or higher.** (Grune and Jacobs 211,212)

Tree-sitter

Tree-sitter is a parser generator tool and **an incremental parsing library**. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited.

Parsing Library – ruby-tree-sitter



- Parsing a string
- Inspect the tree
- Walk the tree
- Visitors – depth-first & breadth-first
- Query the tree

ruby-tree-sitter

- Compiling a parser
- Configuring TreeStand to use the shared object / dynamic library
- Parsing a string
- Inspecting the tree `pp tree`
- Walking the tree, Tree#walk, Tree#each
- Visitors, DepthFirst vs BreadthFirst
  - on, around, on_*, around_*
- Visiting Children (briefly mention `Node#each`)
- Query & Playground

ruby-tree-sitter - compiling a parser

General

```
cc -shared -fPIC -I./src src/parser.c -o parser.so
```

With a “Scanner”

```
cc -shared -fPIC -I./src src/parser.c src/scanner.c -o parser.so
```

On MacOS

```
cc -shared -fPIC -I./src src/parser.c -o parser.dylib
```

With Parser Language

```
cc -shared -fPIC -I./src src/parser.c -o sql.so
```

ruby-tree-sitter - Configuring TreeStand

```
require "tree_stand"

TreeStand.configure do
  config.parser_path = "path/to/parser/folder/"
end

sql_parser = TreeStand::Parser.new("sql")
ruby_parser = TreeStand::Parser.new("ruby")
```

```
This will look for:
path/to/parser/folder/sql.so
```

ruby-tree-sitter - Parsing a String

```
tree = sql_parser.parse_string(<<~SQL) # Mr. Developer
  SELECT u.honorific, r.title
  FROM users u
  JOIN role r
    ON u.id = r.user_id
  WHERE u.name = “Derek”
SQL
```

ruby-tree-sitter - Inspecting the tree

```
pp tree.root_node
```

```
(program
 (statement
  (select
   (keyword_select)                                                             | SELECT
   (select_expression
    (term
     value: (field (object_reference name: (identifier)) name: (identifier)))   | u.honorific
    (term
     value: (field (object_reference name: (identifier)) name: (identifier))))) | r.title
  (from
   (keyword_from)                                                               | FROM
   (relation (object_reference name: (identifier)) alias: (identifier))         | users u
   (join
    (keyword_join)                                                              | JOIN
    (relation (object_reference name: (identifier)) alias: (identifier))        | role r
    (keyword_on)                                                                | ON
    predicate: (binary_expression
     left: (field (object_reference name: (identifier)) name: (identifier))     | u.id
     operator: ("=")                                                            | =
     right: (field (object_reference name: (identifier)) name: (identifier))))  | r.user_id
   (where
    (keyword_where)                                                             | WHERE
    predicate: (binary_expression
     left: (field (object_reference name: (identifier)) name: (identifier))     | u.name
     operator: ("=")                                                            | =
     right: (literal))))))                                                      | "Derek"
```

ruby-tree-sitter - Walking the Tree

```
tree.walk { |node|  pp node }    # Depth-First tree walking
tree.each { |node|  pp node }    # alias for walk

root = tree.root_node

root.walk { |node|  pp node }    # walk subtree beginning at current node.
root.each { |child| pp child }   # iterate over child nodes
```

ruby-tree-sitter - Visitors

```ruby
class CountingVisitor < TreeStand::Visitor
  attr_reader :count

  def initialize(root)
    super(root)
    @count = 0
  end

  def on_predicate(node)
    @count += 1
  end
end

# Initialize a visitor
visitor = CountingVisitor.new(root).visit
# Check the result
visitor.count
# => 2
```

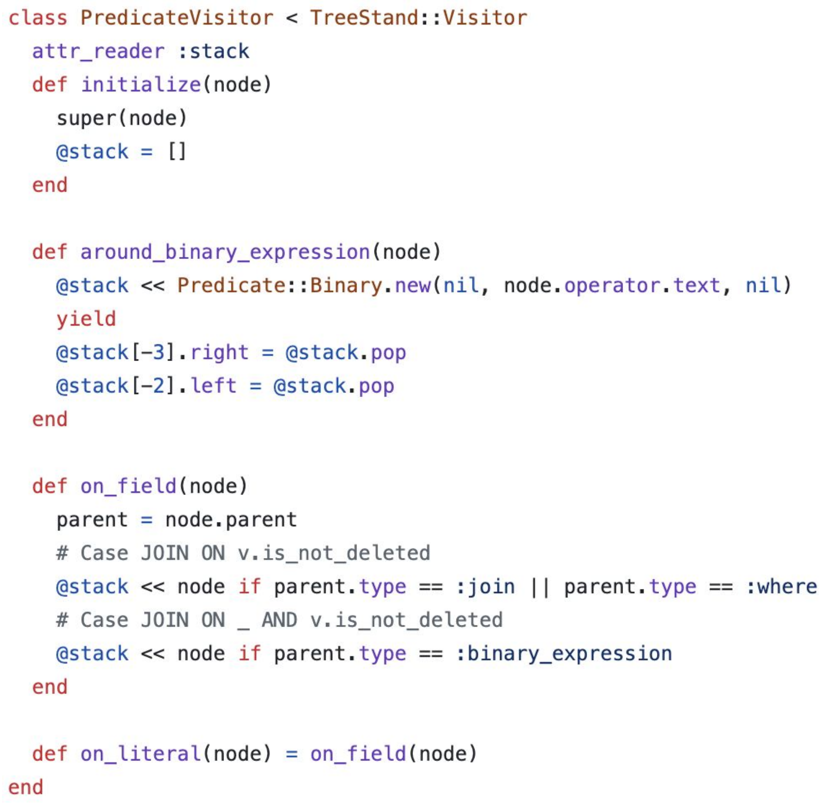
Supports:

```
on(node)
on_[type](node)

around(node, &block)
around_[type](node, &block)
```

ruby-tree-sitter - Visitors (cont)

```ruby
def around(node)
  @stack << TreeNode.new(node, [])
  yield # visit all children of this node

  # The last node on the stack is the root of the tree.
  return if @stack.size == 1

  # Pop the last node off the stack and add it to the parent
  @stack[-2].children << @stack.pop
end
```

github://DerekStride/sql_tools - Visitors (cont)

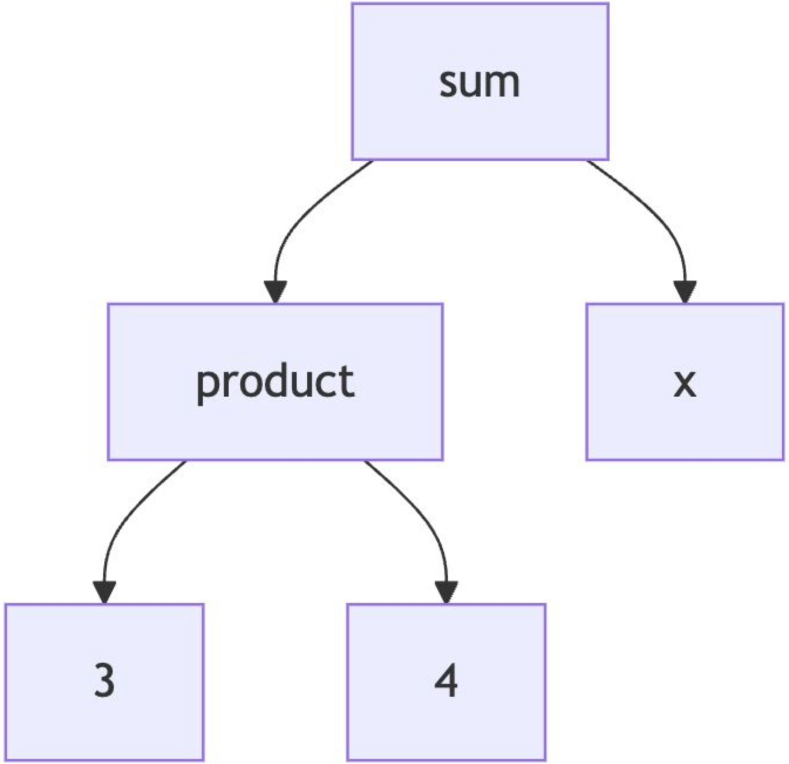


```ruby
class PredicateVisitor < TreeStand::Visitor
  attr_reader :stack
  def initialize(node)
    super(node)
    @stack = []
  end

  def around_binary_expression(node)
    @stack << Predicate::Binary.new(nil, node.operator.text, nil)
    yield
    @stack[-3].right = @stack.pop
    @stack[-2].left = @stack.pop
  end

  def on_field(node)
    parent = node.parent
    # Case JOIN ON v.is_not_deleted
    @stack << node if parent.type == :join || parent.type == :where
    # Case JOIN ON _ AND v.is_not_deleted
    @stack << node if parent.type == :binary_expression
  end

  def on_literal(node) = on_field(node)
end
```

ruby-tree-sitter - Walking the Tree

```
tree.walk { |node|  pp node }    # Depth-First tree walking
tree.each { |node|  pp node }    # alias for walk

node.walk { |walk|  pp walk }    # walk subtree beginning at current node.
node.each { |child| pp child }   # iterate over child nodes
```

Querying the Syntax Tree



```
(product

  left: (literal)
  right: (literal))
```

```
(product

  left: (literal) @left
  right: (literal) @right) @root
```

ruby-tree-sitter - Querying the Syntax Tree

```
tree.query(<<~QUERY)
  (select_expression
    (term
      value:
        (field
          (object_reference name: (identifier) @alias)?
          name: (identifier) @column)))
QUERY

[
  {"alias"=>"u", "column"=>"honorific"},
  {"alias"=>"r", "column"=>"title"},
]
```

```
SELECT u.honorific, r.title
FROM users u
JOIN role r
  ON u.id = r.user_id
WHERE u.name = “Derek”
```

Integration Test

Putting it all together - Compiling the parser

```
git clone github://DerekStride/tree-sitter-server_timing.git

cd tree-sitter-server_timing

cc -shared -fPIC -I./src src/parser.c -o server_timing.dylib
```

Putting it all together - Setup

```
require "tree_stand"

TreeStand.configure do
  config.parser_path = File.join(__dir__, “treesitter”)
end

parser = TreeStand::Parser.new(“server_timing”)
```

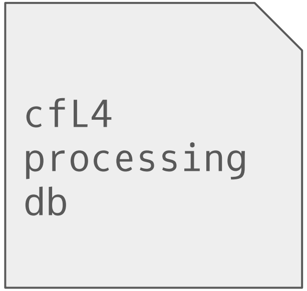
Putting it all together - Parsing a Header

```
require "net/http"


res = Net::HTTP.get_response(URI(“https://derek.stride.host”))
tree = parser.parse_string(res[“Server-Timing”])
pp tree.root_node
```

```
(header
  (timing_metric
    name: (token)
    description: (description value: (string))))
```

Putting it all together - Parsing a Header

```
matches = tree.query("(timing_metric name: (token) @name)")
matches.each do |m|
  puts m["name"].text
end
```



cfL4
processing
db

Putting it all together - Using Queries to Build ruby objects

```ruby
TimingMetric = Data.define(:name, :duration, :description)

matches = tree.query(<<~QUERY).map do |match|
  (timing_metric
    name: (token) @name
    duration: (duration value: (number) @duration)?
    description: (description value: (string) @description)?)
QUERY

  name = match.fetch("name").text
  duration = match["duration"]&.text&.to_f
  description = match["description"]&.text

  TimingMetric.new(name:, duration:, description:)
end
```

Grammar:
github://DerekStride/tree-sitter-server_timing

Ruby Gem:
github://DerekStride/server_timing-ts

Thank You, Parser Generators! 🙏

```
TimingMetric = Data.define(:name, :duration, :description)
Query = <<~QUERY
  (timing_metric
    name: (token) @name
    duration: (duration value: (number) @duration)?
    description: (description value: (string) @description)?)
QUERY

matches = tree.query(Query).map do |match|
  name = match.fetch("name").text
  duration = match["duration"]&.text&.to_f
  description = match["description"]&.text

  TimingMetric.new(name:, duration:, description:)
end
```

Hand–writing a parser

```ruby
header.split(",").map do |raw_metric|
  parts = raw_metric.split(";").map(&:strip)

  dur   = parts.find { |part| part.start_with?("dur=") }
  parts.delete(dur)

  desc  = parts.find { |part| part.start_with?("desc=") }
  parts.delete(desc)

  name  = parts.shift

  duration = dur&.split("=")&.last&.to_f
  description = desc&.split("=")&.last
  TimingMetric.new(name:, duration:, description:)
end
```

Performance Comparison

```
➜ server_timing-ts ruby --yjit bin/bench

ruby 3.3.4 (2024-07-09 revision be1089c8ec) +YJIT [arm64-darwin23]

Warming up --------------------------------------
           generated     1.695k i/100ms
         handwritten     8.195k i/100ms

Calculating -------------------------------------

 generated    16.97k (± 0.7%) i/s  (58.9 μs/i) -   86.445k in  5.092s
 handwritten  81.78k (± 1.7%) i/s  (12.2 μs/i) -  409.750k in  5.011s
```

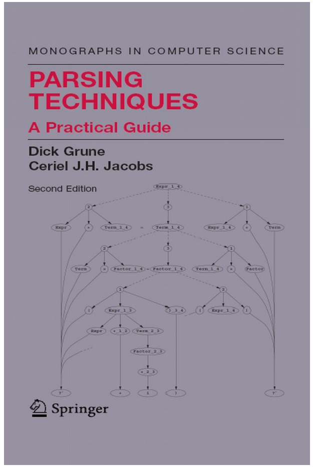
~ 4.75x speed up

Tree-sitter

Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited.

Resources

- Tree-Sitter documentation
- "Tree-sitter - a new parsing system for programming tools" by Max Brunsfeld
- derek.stride.host/posts/comprehensive-introduction-to-tree-sitter
- github://Faveod/ruby-tree-sitter
- YARD documentation
- Parsing Techniques - A Practical Guide
- Dragon Book



MONOGRAPHS IN COMPUTER SCIENCE

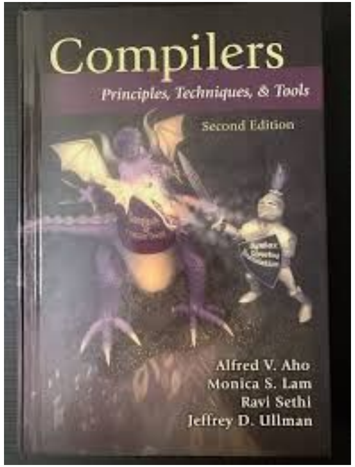
PARSING TECHNIQUES

A Practical Guide

Dick Grune
Ceriel J.H. Jacobs

Second Edition

Springer



Compilers

Principles, Techniques, & Tools

Second Edition

Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

Questions?