たのしいparse.y

2024/12/05 RubyWorld Conference 2024

@ydah

株式会社アンドパッド 開発本部 SWE

わたしについて

Yudai Takada (@ydah)

開発本部

SWE

Profile | **経 歴**

大学卒業後、ソフトウェアの受託開発会社を経て、株式会社アンドパッドに入社。現在はリアーキテクティングチームに所属、大規模なRailsアプリケーションのリアーキテクチャや横断的な技術課題の解決に取り組んでいる。生まれも育ちも大阪で、大阪の地域Rubyコミュニティであるkyobashi.rbの創設メンバー。また、大阪Ruby会議04ではチーフオーガナイザーを務めた。近年は、Rubyのパーサージェネレーター「Lrama」のコミッターとして、Rubyの構文解析器の改善に取り組んでいる。

本日が2024ツアーファイナルです

- 🇯🇵 2024.05.17 (Naha) RubyKaigi 2024
- 🇯🇵 2024.08.24 (Osaka) OsakaRubyKaigi04 (Organizing)
- 🌏 2024.08.31 (ONLINE) RubyKaigi 2024 followup
- 🇧🇦 2024.09.13 (Sarajevo) EuRuKo 2024
- 🇯🇵 2024.10.26 (Tokyo) Kaigi on Rails 2024
- 🇯🇵 ***2024.12.05 (Matsue) RubyWorld Conference2024***

来年の6月は京都で会いましょう

関西Ruby会議08 / KansaiRubyKaigi08

June 2025 in Kyoto

ANDPADとは

現場の効率化から経営改善まで一元管理できる

クラウド型建設プロジェクト管理サービス

社　内

営業 / 監督 / 設計
事務 / 管理職

ANDPAD

社　外

職人 / 業者
メーカー / 流通

案件管理　資料　工程表　写真　報告　チャット　黒板　図面　受発注　・・・

本日のスライド

本日のトークのゴール

1. **parse.y を取り巻く要素を知る**

   レキサー、パーサー、パーサージェネレーターってなんなのさ

2. **parse.y を読み解くときの取っ掛かりを得る**

   ファイルの一番上から読み進めるなんてとんでもない！

3. **parse.y のたのしい歩き方を知る**

   悪魔城や魔境や地獄と呼ばれることはありますが...

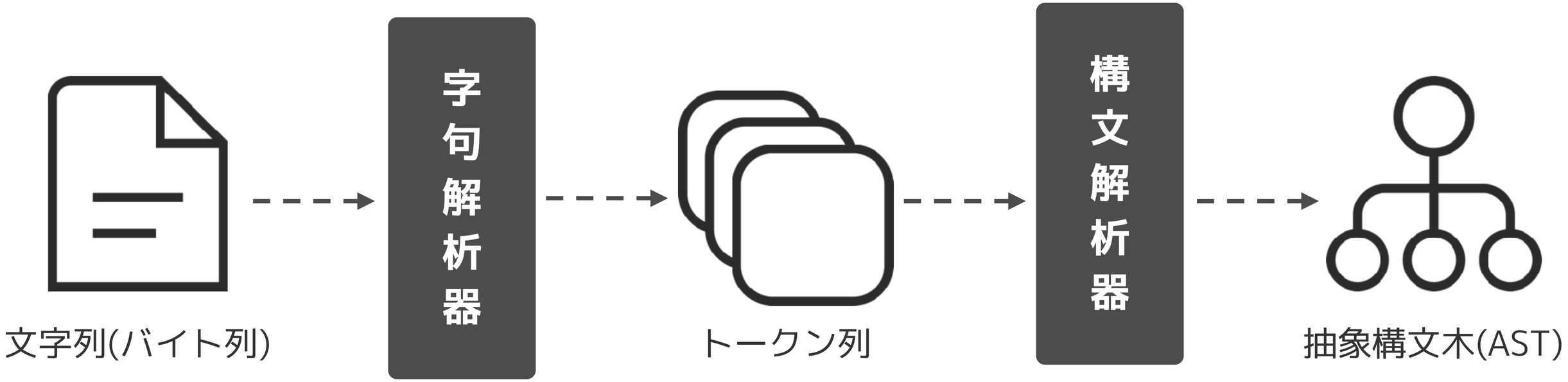
01

イントロダクション - parse.yとその周辺

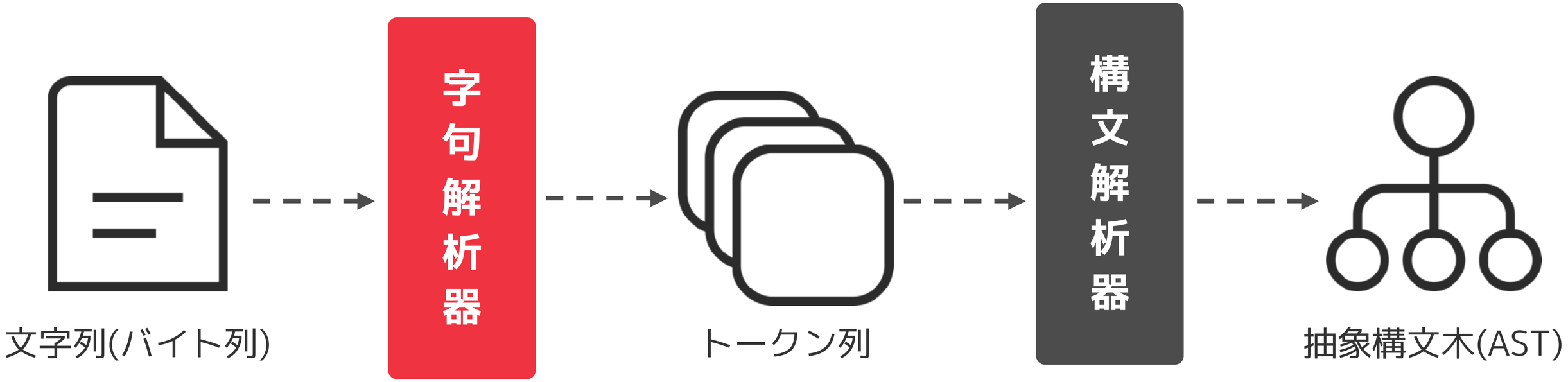
"この世のどんなものもみな「初めて」から出発するのだから"

——高階杞一「準備」

構文解析器(パーサー)と字句解析器(レキサー)



構文解析器(パーサー)と字句解析器(レキサー)



字句解析器(レキサー)の役割

文字の並びを解析し、

意味のある最小の単位（トークン）に分解する

```
def method_name(param)
  puts param
end
```

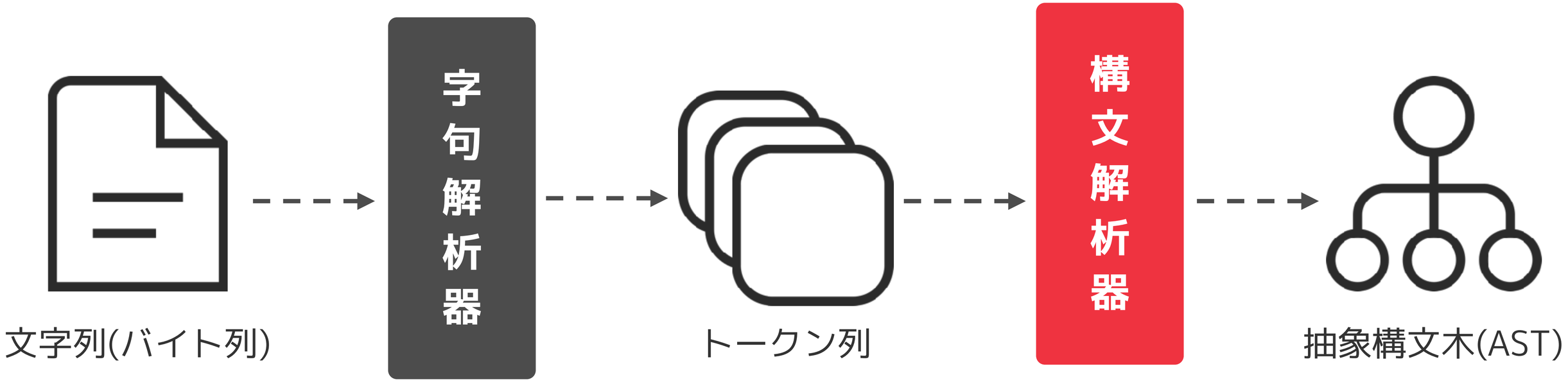
字句解析器(レキサー)の役割

文字の並びを解析し、

意味のある最小の単位（トークン）に分解する

```
def method_name(param)
  puts param
end
```

構文解析器(パーサー)と字句解析器(レキサー)

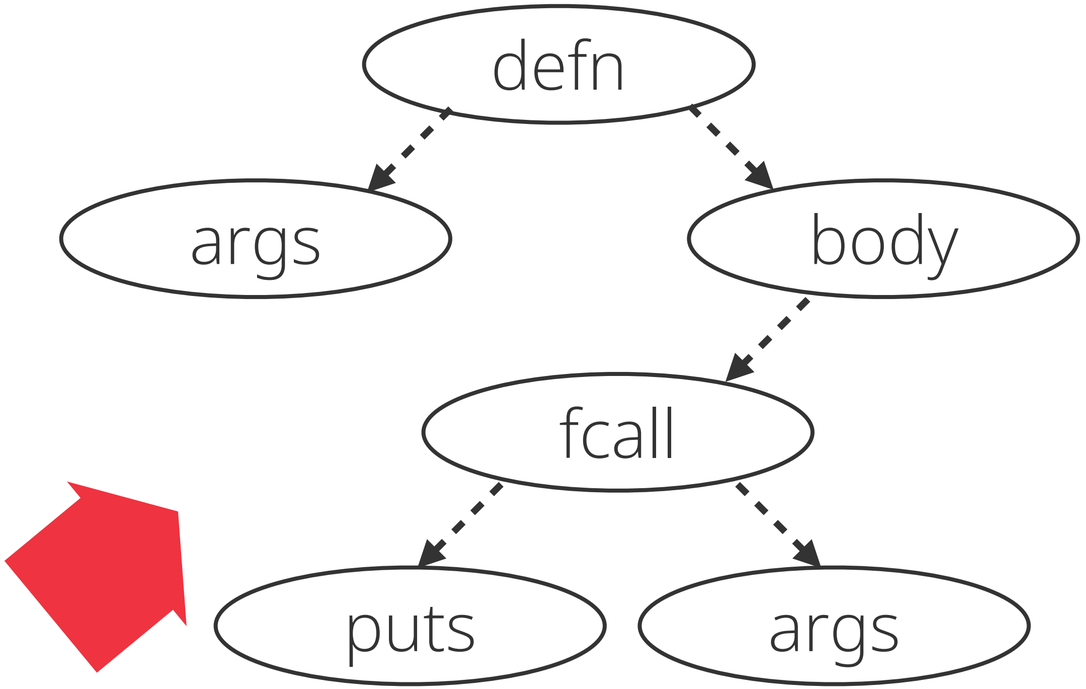


構文解析器(パーサー)の役割

字句解析から受け取ったトークン列が

文法的に正しいかを検査して抽象構文木(AST)を作成する

```
def method_name(param)
  puts param
end
```



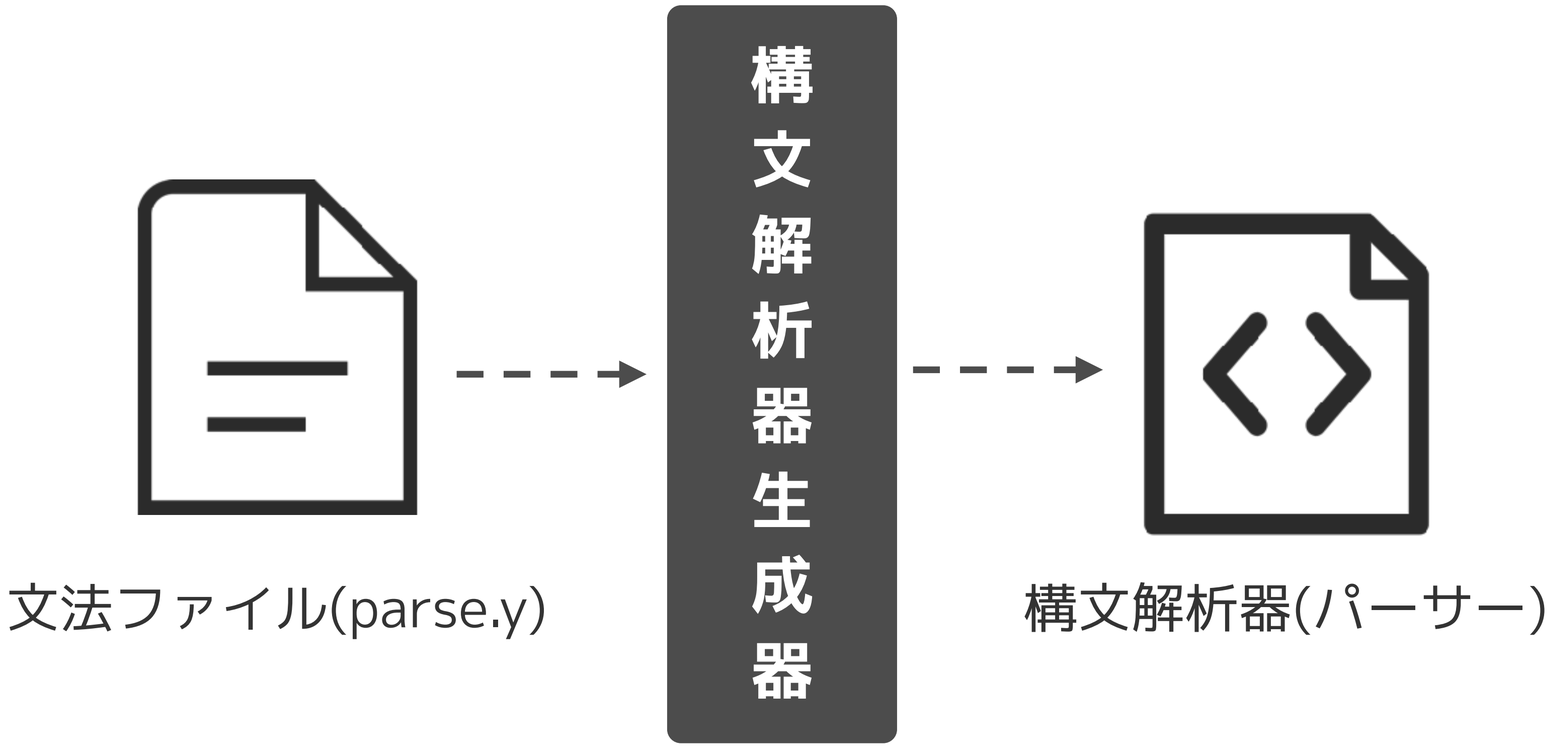
keyword_def | tIDENTIFIER | '(' | tIDENTIFIER | ')' | ...

構文解析器(パーサー)がASTを生成する様子を見る

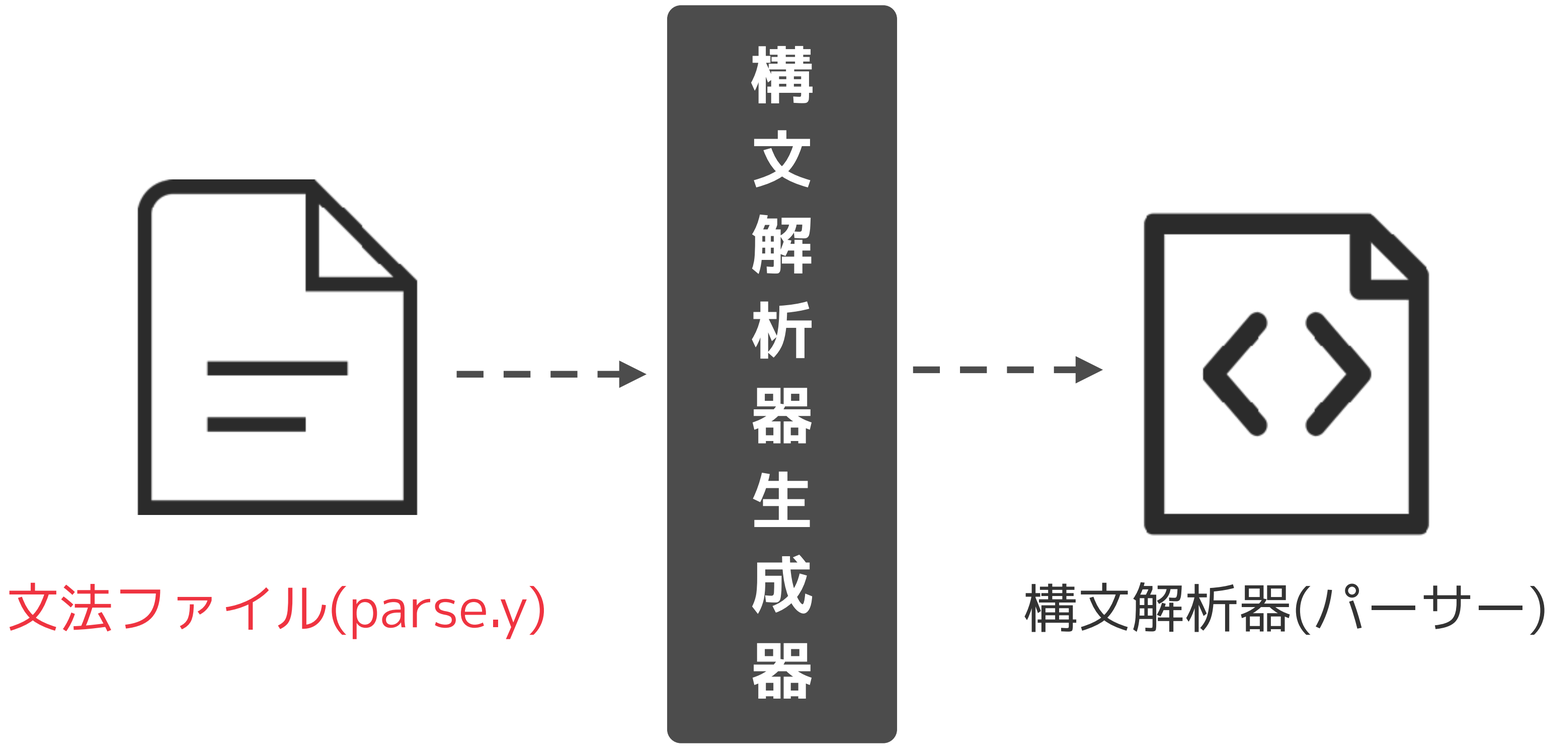
ruby --parser=parse.y --dump=parsetree -e "code"

```
❯ ruby --parser=parse.y --dump=parsetree -e "p 'たのしいparse.y'"
: (snip)
# @ NODE_SCOPE (id: 3, line: 1, location: (1,0)-(1,23))
# +- nd_tbl: (empty)
# +- nd_args:
# |   (null node)
# +- nd_body:
#     @ NODE_FCALL (id: 0, line: 1, location: (1,0)-(1,23))*
#     +- nd_mid: :p
#     +- nd_args:
#         @ NODE_LIST (id: 2, line: 1, location: (1,2)-(1,23))
#         +- as.nd_alen: 1
#         +- nd_head:
#         |   @ NODE_STR (id: 1, line: 1, location: (1,2)-(1,23))
#         |   +- string: "たのしいparse.y"
#         +- nd_next:
```

parse.yと構文解析器生成器(パーサージェネレーター)



parse.yと構文解析器生成器(パーサージェネレーター)



文法ファイル(parse.y)

構文解析器の設計図

Ruby のプログラムを構文解析し、抽象構文木(AST)を生成するために必要な規則や記号が定義されているファイル。

Ruby 3.2以前ではGNU Bison、Ruby 3.3 以降では Lramaというパーサージェネレータを使ってコンパイルされる。

Ruby の柔軟で複雑な文法は、このparse.yというRubyの文法の設計図によって実現されている。Ruby は実質 parse.y (!)

ここまでのまとめ

parse.yを取り巻く要素

- 字句解析器(レキサー)は文字列をトークンに変換する役割を担う
- 構文解析器(パーサー)はトークン列が文法的に正しいかをチェックして抽象構文木(AST)を作成する役割を担う
- parse.y はパーサージェネレーターが構文解析器(パーサー)を作成するための文法ファイル

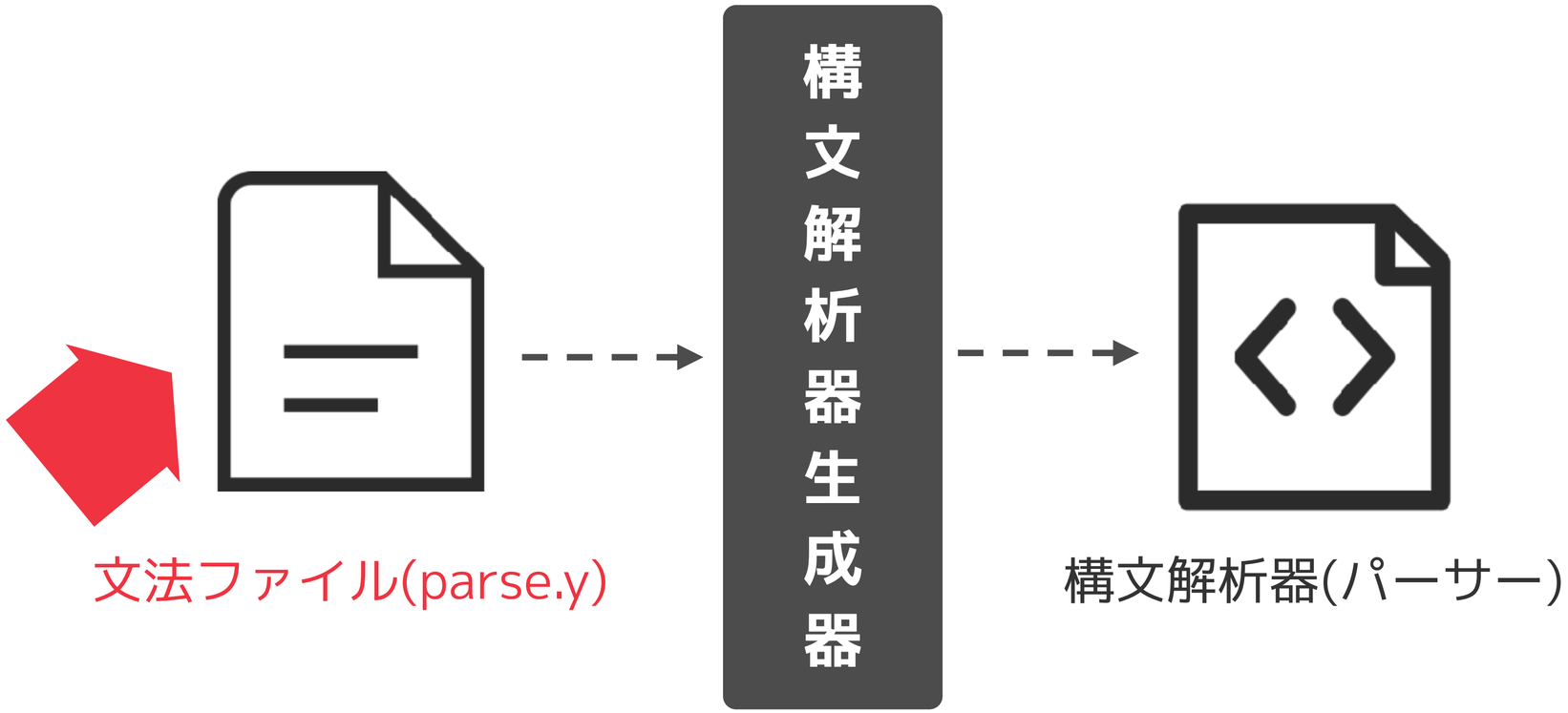
02

parse.yをはじめよう

"自分の皮膚で触れた部分が世界なんですよ"

——伊坂幸太郎「砂漠」

ここからは文法ファイル(parse.y)の中の世界へ



文法

Backus–Naur Form(BNF)

```
expr : NUMBER { $$ = $1; }
     | expr '+' expr { $$ = $1 + $3; }
     ;
```

Backus–Naur Form(BNF)とは

文脈自由文法(Context-free Grammar)を定義するのに使うメタ言語。

John Warner BackusとPeter NaurがALGOL 60というプログラミング言語の文法定義のために考案し1959年に論文を発表した。

Backus, J.W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. IFIP Congress.

https://api.semanticscholar.org/CorpusID:44764020

足し算を表すBNFの例

```
%token <int> NUMBER
%%
expr : NUMBER { $$ = $1; }
     | expr '+' expr { $$ = $1 + $3; }
     ;
```

文法

左辺が文法規則の名前、右辺は展開する規則

LHS (Left Hand Side)

```
%token <int> NUMBER
%%
expr : NUMBER { $$ = $1; }
     | expr '+' expr { $$ = $1 + $3; }
     ;
```

RHS (Right Hand Side)

終端記号と非終端記号

```
%token <int> NUMBER
%%
expr : NUMBER { $$ = $1; }
     | expr '+' expr { $$ = $1 + $3; }
     ;
```

終端記号 (Terminal Symbol)

終端記号と非終端記号

```
%token <int> NUMBER
%%
expr : NUMBER { $$ = $1; }
     | expr '+' expr { $$ = $1 + $3; }
     ;
```

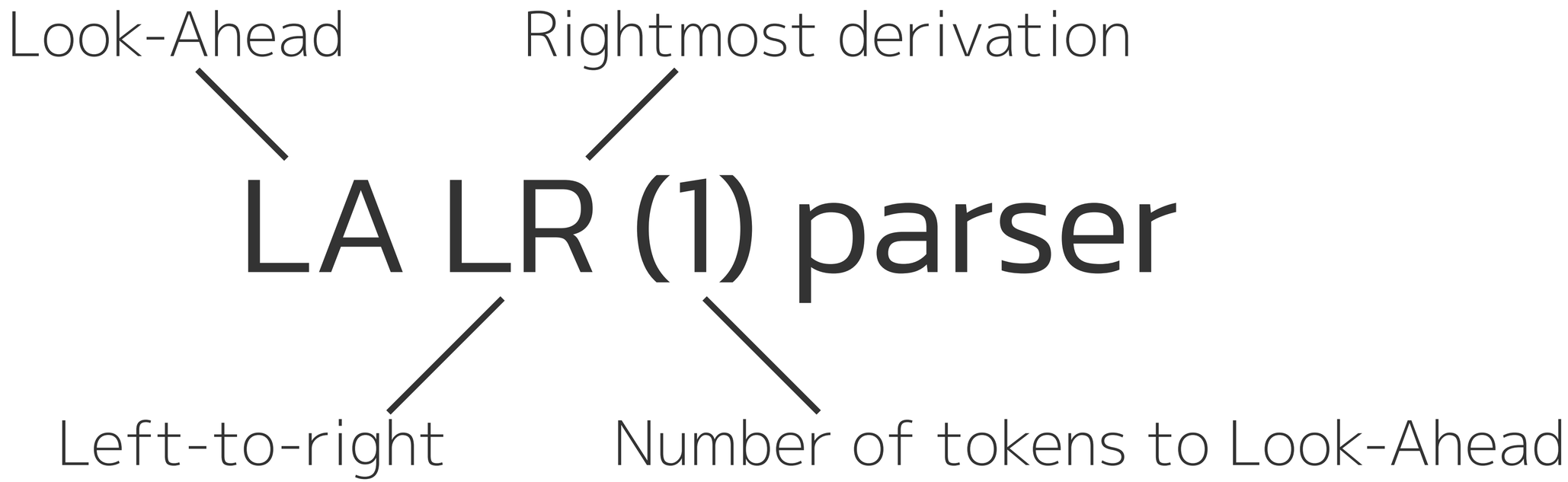
非終端記号 (Nonterminal Symbol)

LR法における解析：シフト(shift)と還元(reduce)

「シフト(shift)」と「還元(reduce)」は、LR構文解析の基本操作。これらを使ってLR構文解析器は入力を解析する。

- シフト: 入力から次のトークンを読み取り、構文解析器のスタックに積む操作
- 還元: スタック上に積まれているトークンの並びが、ある文法規則に一致するときに、一つの非終端記号に置き換える操作

Lrama: Pure Ruby LALR parser generator



文法の解析方法

最もシンプルな足し算の例

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```

スタック

文法の解析方法

左辺の数値がレキサーから渡される

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```

文法の解析方法

+ がレキサーから渡される

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```

文法の解析方法

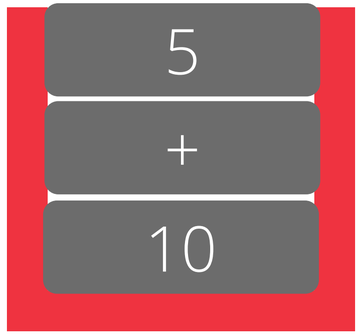
右辺の数値がレキサーから渡される

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```

文法の解析方法

文法規則に一致したので還元

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```



スタック

文法の解析方法

一つの非終端記号に置き換える

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```

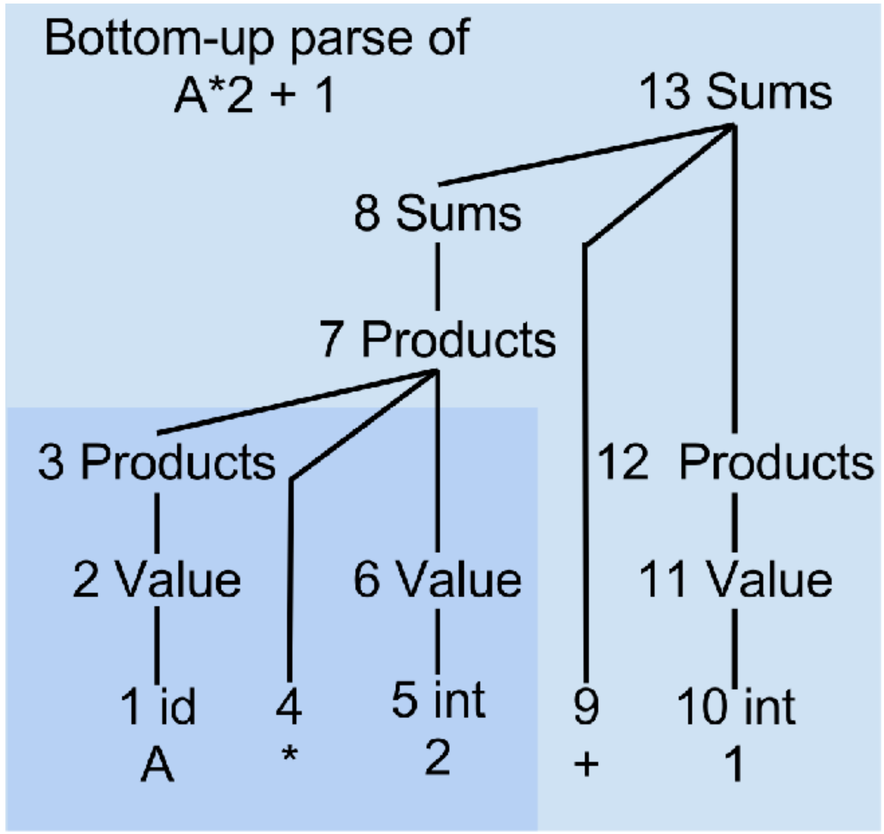
スタック

LR構文解析器はどのように抽象構文木(AST)を作るのか

ボトムアップで(葉から順に)組み上げていく

BNFも、左辺が親、右辺が子として木構造で表現される。

LR構文解析器はシフトを行いながら、葉から順に還元を進め、抽象構文木（AST）を構築していく。



Bottom-up parse tree built in numbered steps

https://en.wikipedia.org/wiki/LR_parser#/media/File:Shift-Reduce_Parse_Steps_for_A*2+1.svg

BisonやLramaの抽象構文木構築のアプローチ

還元する瞬間をフックするアクションでASTを作成する

(Semantic) Actions

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```

抽象構文木を作るのに必要なこと

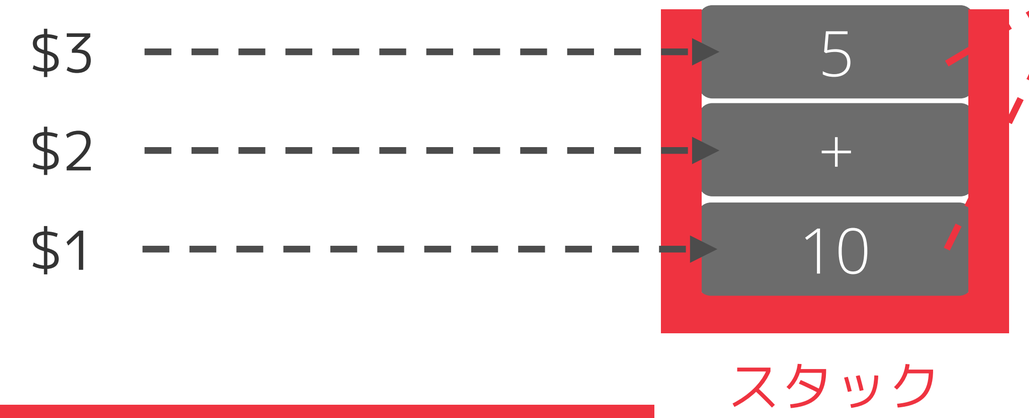
終端/非終端記号の値へのアクセス

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```

抽象構文木を作るのに必要なこと

終端/非終端記号の値へのアクセス

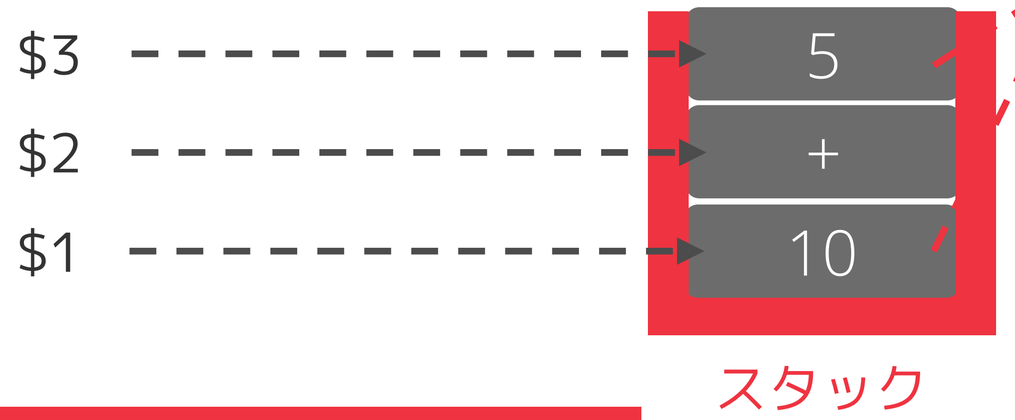
```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```



抽象構文木を作る

親へアクションの結果を渡す

```
%token <int> NUMBER
%%
expr : NUMBER '+' NUMBER { $$ = $1 + $3; }
     ;
```



抽象構文木を作るのに必要なこと

親へアクションの結果を渡す

```
expr : sum { $$ = $1; };

sum  : NUMBER '+' NUMBER { $$ = $1 + $3; };
```

抽象構文木を作るのに必要なこと

親へアクションの結果を渡す

```
expr : sum

       { $$ = $1; };

       sum  : NUMBER '+' NUMBER

       { $$ = $1 + $3; };
```

抽象構文木を作るのに必要なこと

親へアクションの結果を渡す

```
expr : sum

       { $$ = $1; };

       sum  : NUMBER '+' NUMBER

       { $$ = $1 + $3; };
```

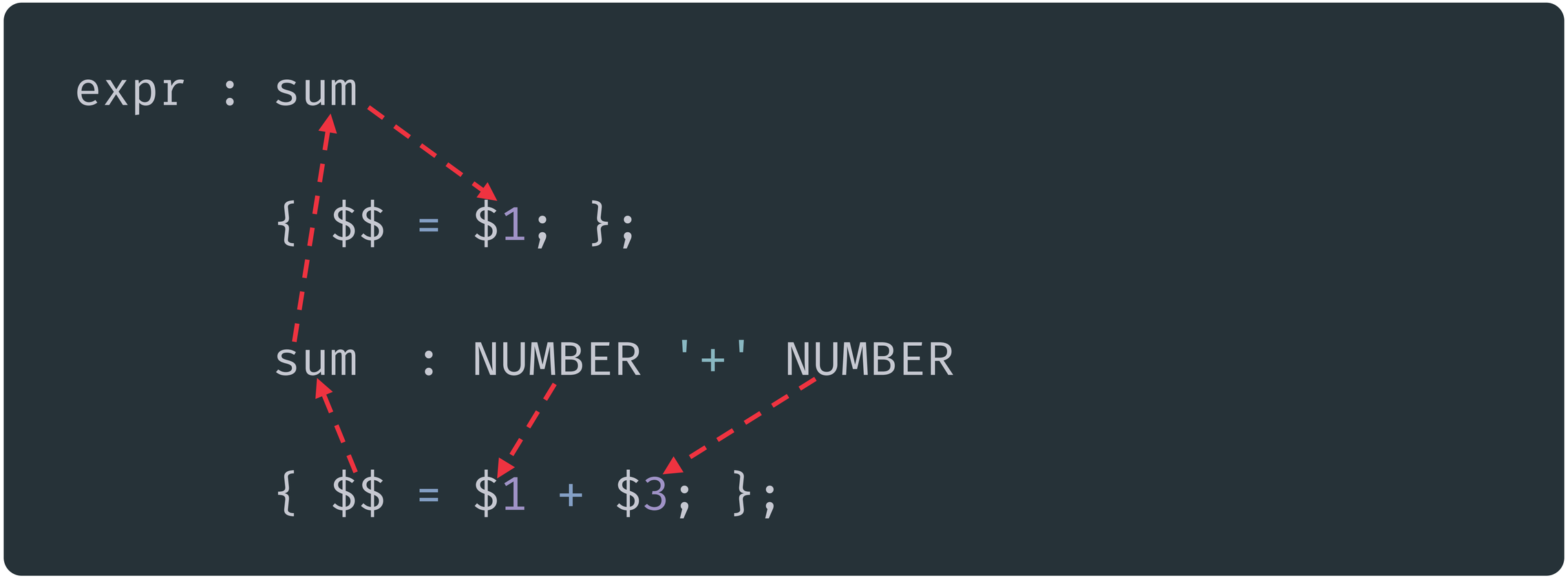
抽象構文木を作るのに必要なこと

親へアクションの結果を渡す

```
expr : sum
       { $$ = $1; };

sum  : NUMBER '+' NUMBER
       { $$ = $1 + $3; };
```

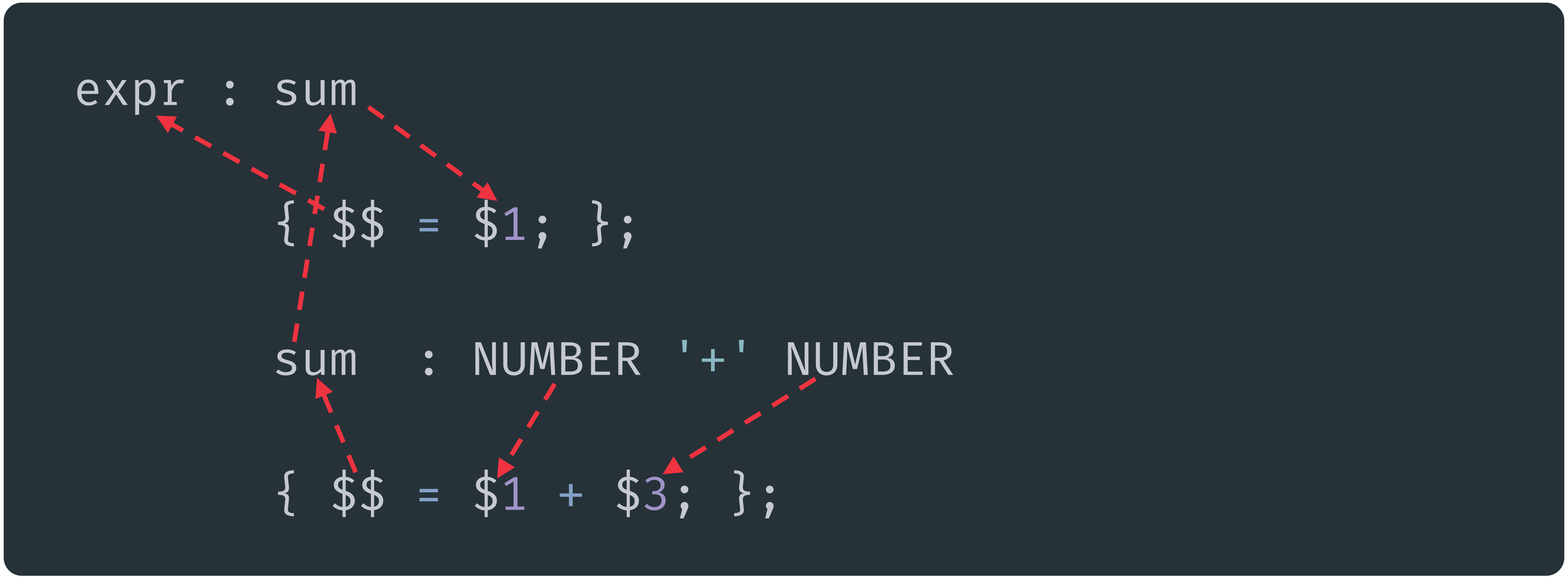
抽象構文木を作るのに必要なこと

親へアクションの結果を渡す



抽象構文木を作るのに必要なこと

親へアクションの結果を渡す



ここまでのまとめ

parse.yの構造と文法

- parse.yはBackus–Naur Form(BNF)を用いて書かれている
- 文法の解析にはシフト(shift)と還元(reduce)を使用して入力を解析する
- LR構文解析器はシフトを行いながら、葉から順に還元を進め、抽象構文木(AST)を構築する
- 還元時に実行されるアクション内で抽象構文木(AST)を作成する

03

読み解くときのポイント

"旅人よ、道はない。 歩くことで道はできる"

ーーアントニオ・マチャド「カスティーリャの大地」

プログラムの海を航海するためには

地図とコンパスが必要

(さもなくばあてもなく漕ぎ続けることになる)

プログラムの海を航海する道具

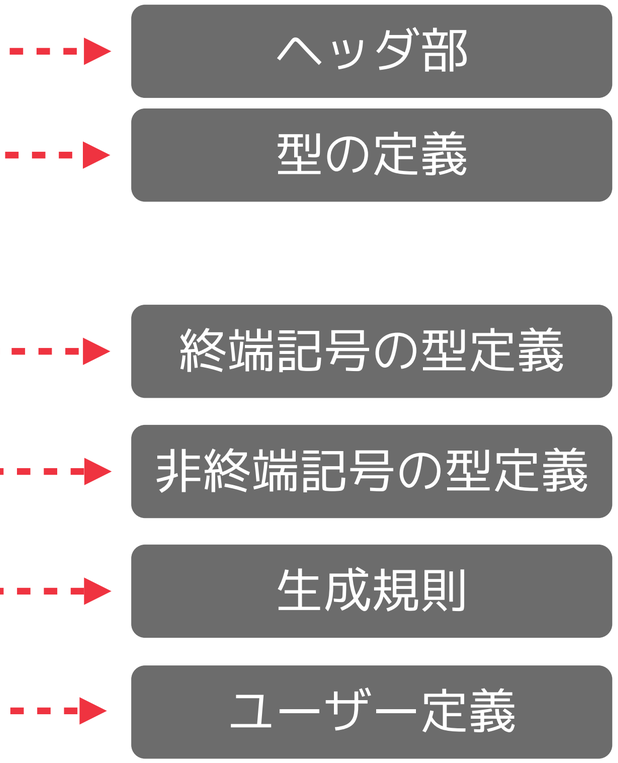
parse.yの地図を手に取り

地理的な位置関係を理解する

文法ファイルの構造を知る

大まかなファイルの構造

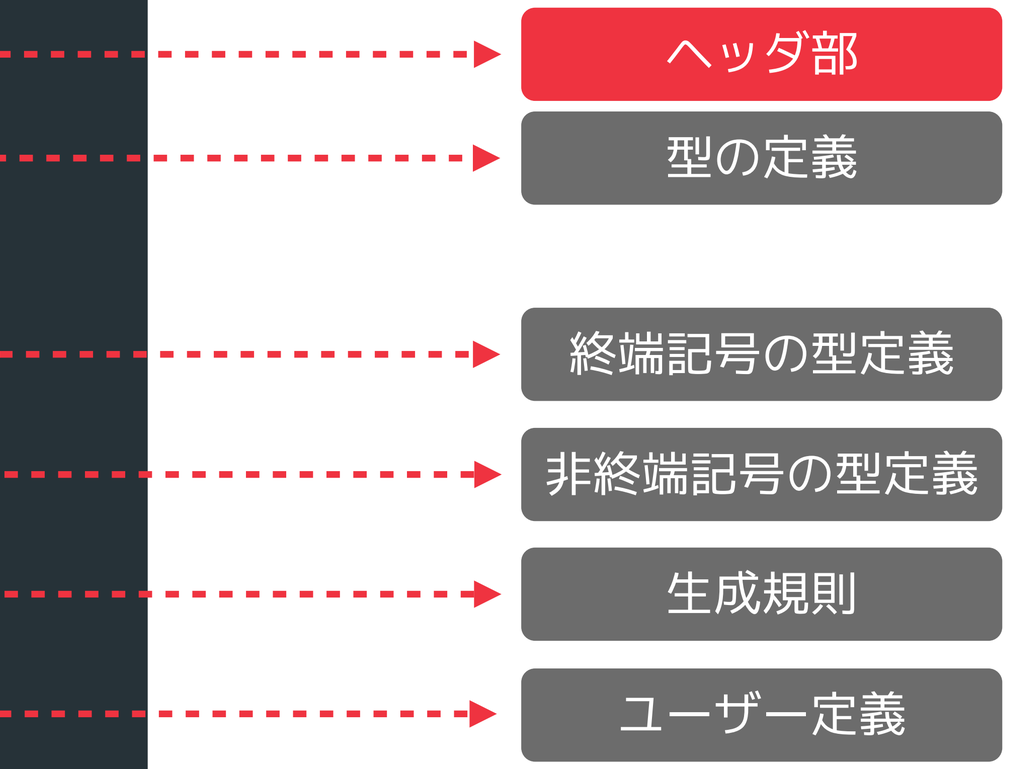
```
%{
    // C code
%}
%union {
    NODE *node;
     …
}
%token <id> keyword_class "'class'"
 …
%type <node> singleton strings  …
%%
program    : top_compstmt
 …
%%
// C code
```



文法ファイルの構造を知る

大まかなファイルの構造

```
%{
    // C code
%}
%union {
    NODE *node;
     …
}
%token <id> keyword_class "'class'"
 …
%type <node> singleton strings  …
%%
program    : top_compstmt
 …
%%
// C code
```



ヘッダ部

必要なヘッダをインクルードしていたり、

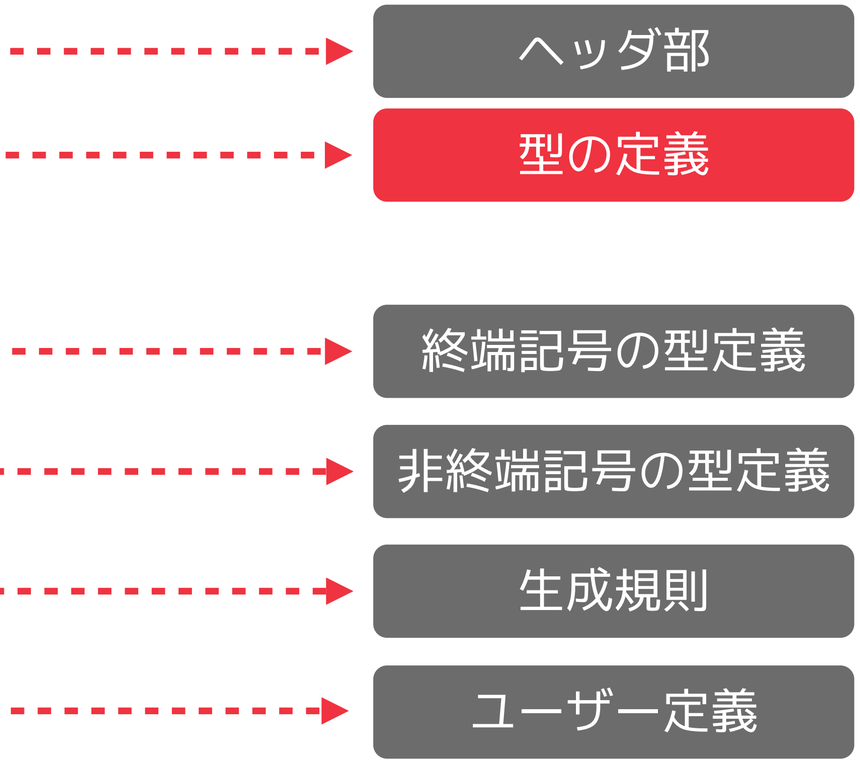
マクロを定義していたり、データ構造を定義していたり

```
#include "internal.h"
:
:
#define RUBY_SET_YYLLOC_FROM_STRTERM_HEREDOC(Current)           \
    rb_parser_set_location_from_strterm_heredoc(p, &p->lex.strterm->u.heredoc, &(Current))
:
:
typedef struct parser_string_buffer_elem {
    struct parser_string_buffer_elem *next;
    long len;  /* Total length of allocated buf */
    long used; /* Current usage of buf */
    rb_parser_string_t *buf[FLEX_ARY_LEN];
} parser_string_buffer_elem_t;
```

文法ファイルの構造を知る

大まかなファイルの構造

```
%{
    // C code
%}
%union {
    NODE *node;
     …
}
%token <id> keyword_class "'class'"
 …
%type <node> singleton strings  …
%%
program    : top_compstmt
 …
%%
// C code
```



型の定義

構文解析中に使えるさまざまな型の値を
保持できる共用体を作成する

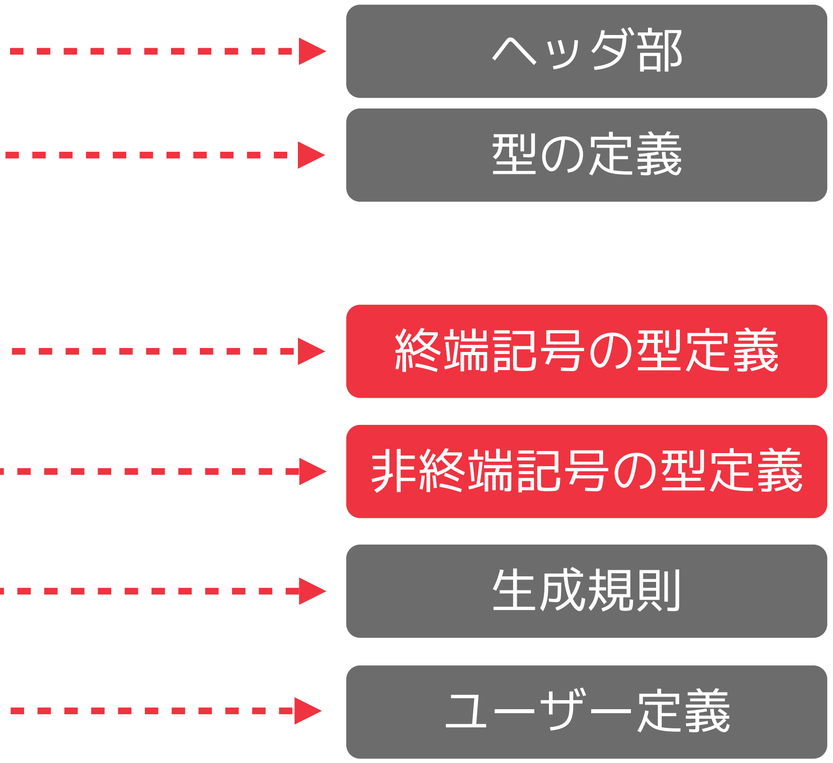
```
%union {
    NODE *node;
    ID id;
    int num;
}
```

```
union YYSTYPE
{
#line 25 "sample/calc.y"

    NODE *node;
    ID id;
    int num;

#line 134 "y.tab.c"
};
typedef union YYSTYPE YYSTYPE;
```

文法ファイルの構造を知る

大まかなファイルの構造

```
%{
    // C code
%}
%union {
    NODE *node;
     …
}
%token <id> keyword_class "'class'"
 …
%type <node> singleton strings  …
%%
program    : top_compstmt
 …
%%
// C code
```



記号の型定義

終端記号は %token を使って

非終端記号は %type を使って定義する

```
%token <id>   tIDENTIFIER    "local variable or method"
%token <id>   tFID           "method"
%token <id>   tGVAR          "global variable"
%token <id>   tIVAR          "instance variable"
%token <id>   tCONSTANT      "constant"
%token <id>   tCVAR          "class variable"
%token <id>   tLABEL         "label"
 ...

%type <node> singleton strings string string1 xstring regexp
%type <node> string_contents xstring_contents regexp_contents string_content
%type <node> words symbols symbol_list qwords qsymbols word_list qword_list qsym_list word
 ...
```

終端記号の型定義

%token を使用して終端記号と型情報を紐づける

```
%token <id>    tIDENTIFIER    "local variable or method"
%token <id>    tFID           "method"
%token <id>    tGVAR          "global variable"
%token <id>    tIVAR          "instance variable"
%token <id>    tCONSTANT      "constant"
%token <id>    tCVAR          "class variable"
%token <id>    tLABEL         "label"
```

型情報　終端記号名　説明用の名前(エラーメッセージ)

非終端記号の型定義

%type を使用して非終端記号と型情報を紐づける

```
%type <node> singleton strings string
%type <node> string1 xstring regexp
%type <node> string_contents xstring_contents
%type <node> regexp_contents string_content
%type <node> words symbols symbol_list qwords
%type <node> qsymbols word_list qword_list qsym_list word
```

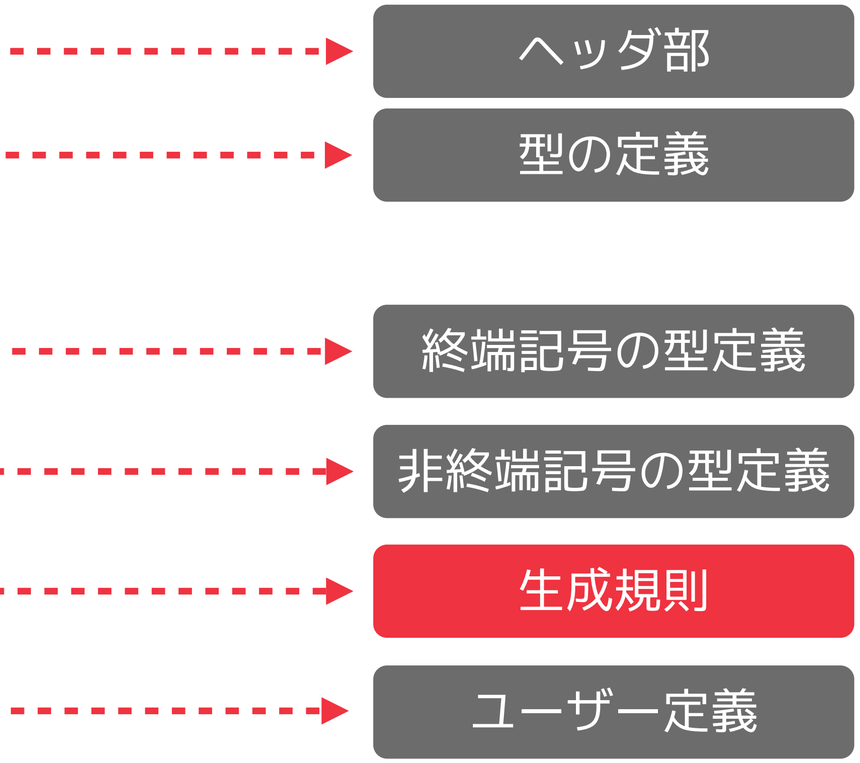
型情報

非終端記号名

文法ファイルの構造を知る

大まかなファイルの構造

```
%{
    // C code
%}
%union {
    NODE *node;
     …
}
%token <id> keyword_class "'class'"
 …
%type <node> singleton strings  …
%%
program    : top_compstmt
 …
%%
// C code
```



生成規則

BNF形式で抽象構文木(AST)を組み立てる

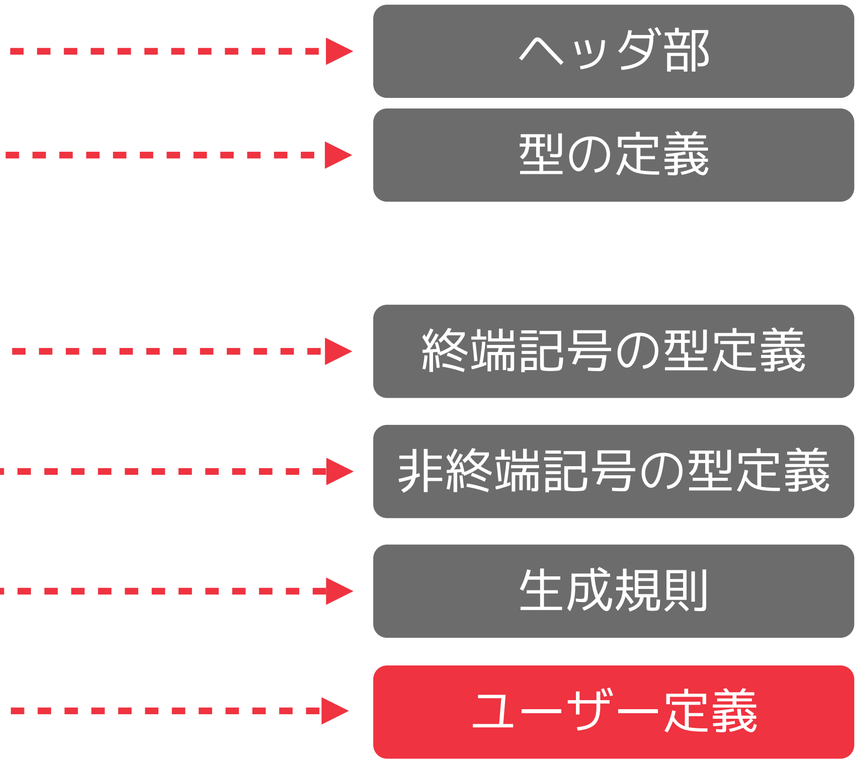
```
block_arg    : tAMPER arg_value
                    {
                        $$ = NEW_BLOCK_PASS($2, &@$, &@1);
                    /*% ripper: $:2 %*/
                    }
             | tAMPER
                    {
                        forwarding_arg_check(p, idFWD_BLOCK, idFWD_ALL, "block");
                        $$ = NEW_BLOCK_PASS(NEW_LVAR(idFWD_BLOCK, &@1), &@$, &@1);
                    /*% ripper: Qnil %*/
                    }
             ;
```

抽象構文木の作成

文法ファイルの構造を知る

大まかなファイルの構造

```
%{
    // C code
%}
%union {
    NODE *node;
     …
}
%token <id> keyword_class "'class'"
 …
%type <node> singleton strings  …
%%
program    : top_compstmt
 …
%%
// C code
```



ユーザー定義

字句解析のための関数yylex()の定義

```
static enum yytokentype
yylex(YYSTYPE *lval, YYLTYPE *yylloc, struct parser_params *p)
{
    enum yytokentype t;

    p->lval = lval;
    lval->node = 0;
    p->yylloc = yylloc;

    t = parser_yylex(p);

    if (has_delayed_token(p))
        dispatch_delayed_token(p, t);
    else if (t != END_OF_INPUT)
        dispatch_scan_event(p, t);

    return t;
}
```

ユーザー定義

抽象構文木(AST)を作成するための関数定義

```
static rb_node_defn_t *
rb_node_defn_new(struct parser_params *p, ID nd_mid, NODE *nd_defn, const YYLTYPE *loc)
{
    rb_node_defn_t *n = NODE_NEWNODE(NODE_DEFN, rb_node_defn_t, loc);
    n->nd_mid = nd_mid;
    n->nd_defn = nd_defn;

    return n;
}
```

プログラムの海を公開する道具

コンパスを使って現在地を把握し進む

parse.yを読む前の準備

慣れていればよいが、一般的には読みにくい(はず)

```
begin_block: block_open top_compstmt '}'
                    {
                        restore_block_exit(p, $block_open);
                        p->eval_tree_begin = block_append(p, p->eval_tree_begin,
                                                          NEW_BEGIN($2, &@$));
                        $$ = NEW_BEGIN(0, &@$);
                    /*% ripper: BEGIN!($:2) %*/
                    }
                ;
```

parse.yを読む前の準備

シンタックスハイライトがあると読みやすくなる

```
begin_block: block_open top_compstmt '}'
                    {
                        restore_block_exit(p, $block_open);
                        p->eval_tree_begin = block_append(p, p->eval_tree_begin,
                                                          NEW_BEGIN($2, &@$));
                        $$ = NEW_BEGIN(0, &@$);
                    /*% ripper: BEGIN!($:2) %*/
                    }
                ;
```

Lrama for Visual Studio Code

github.com/ydah/lrama-for-vscode

解析している様子を見る

ruby --parser=parse.y --yydebug -e "code"

```
❯ ruby --parser=parse.y --yydebug -e "p 'たのしいparse.y"
add_delayed_token:7790 (0: 0|0|0)
Starting parse
Entering state 0
Stack now 0
Reducing stack by rule 1 (line 2971):
lex_state: NONE → BEG at line 2972
vtable_alloc:14925: 0×0000600000f999c0
vtable_alloc:14926: 0×0000600000f999e0
cmdarg_stack(push): 0 at line 14940
cond_stack(push): 0 at line 14941
→ $$ = nterm $@1 (1.0-1.0: )
Entering state 1
Stack now 0 1
Reading a token
lex_state: BEG → CMDARG at line 10545
parser_dispatch_scan_event:11315 (1: 0|1|23)
Next token is token "local variable or method" (1.0-1.1: p)
Shifting token "local variable or method" (1.0-1.1: p)
```

実際の例

https://bugs.ruby-lang.org/issues/20790

Bug #20790 OPEN

Edit ★ Unwatch ♥ Like 0 ...

Syntax acceptance of `*x = p rescue p 1` is different between parse.y and prism

Added by tompng (tomoya ishida) about 2 months ago. Updated 8 days ago.

Status:	Open		
Assignee:	-		
Target version:	-		
ruby -v:	ruby 3.4.0dev (2024-10-09T03:27:05Z master ed11a244dd) +PRISM [x86_64-linux]	**Backport:**	3.1: UNKNOWN, 3.2: UNKNOWN, 3.3: UNKNOWN

[ruby-core:119488]

Description

```
*x = p rescue p 1 # syntax error in prism
*x = p 1 rescue p 1 # both syntax ok
x = p rescue p 1 # both syntax error
x = p 1 rescue p 1 # both syntax ok
```

Which is correct? If `*x = p rescue p 1` is syntax valid, should `x = p rescue p 1` also be syntax valid?

```
*x = p rescue p 1
⇒(*x = p) rescue (p 1)
```

実際の例

--yydebug の結果を眺める

```
❯ ruby --parser=parse.y -ye "*x = p rescue p 1"
: (snip)
Reducing stack by rule 40 (line 3222):
   $1 = nterm mlhs (1.0-1.2: )
   $2 = token '=' (1.3-1.4: )
   $3 = nterm lex_ctxt (1.4-1.4: )
   $4 = nterm mrhs_arg (1.5-1.6: NODE_VCALL)
   $5 = token "`rescue' modifier" (1.7-1.13: )
   $6 = nterm after_rescue (1.13-1.13: )
   $7 = nterm stmt (1.14-1.17: NODE_FCALL)
→ $$ = nterm stmt (1.0-1.17: NODE_MASGN)
```

行情報

実際の例

該当する生成規則を割り出す

```
| mlhs '=' lex_ctxt mrhs_arg modifier_rescue
  after_rescue stmt[resbody]
    {
        p->ctxt.in_rescue = $3.in_rescue;
        YYLTYPE loc = code_loc_gen(&@modifier_rescue, &@resbody);
        $resbody = NEW_RESBODY(0, 0, remove_begin($resbody), 0, &loc);
        loc.beg_pos = @mrhs_arg.beg_pos;
        $mrhs_arg = NEW_RESCUE($mrhs_arg, $resbody, 0, &loc);
        $$ = node_assign(p, (NODE *)$mlhs, $mrhs_arg, $lex_ctxt, &@$);
    /*% ripper: massign!($:1, rescue_mod!($:4, $:7)) %*/
    }
```

実際の例

作成していそうな抽象構文木のNodeを確認する

```
  | mlhs '=' lex_ctxt mrhs_arg modifier_rescue
    after_rescue stmt[resbody]
      {
          p->ctxt.in_rescue = $3.in_rescue;
          YYLTYPE loc = code_loc_gen(&@modifier_rescue, &@resbody);
          $resbody = NEW_RESBODY(0, 0, remove_begin($resbody), 0, &loc);
          loc.beg_pos = @mrhs_arg.beg_pos;
          $mrhs_arg = NEW_RESCUE($mrhs_arg, $resbody, 0, &loc);
          $$ = node_assign(p, (NODE *)$mlhs, $mrhs_arg, $lex_ctxt, &@$);
      /*% ripper: massign!($:1, rescue_mod!($:4, $:7)) %*/
      }
```

実際の例

作成されるASTと突き合わせる

```
❯ ruby --parser=parse.y --dump=parsetree -e "*x = p rescue p 1"
:
# +- nd_body:
#    @ NODE_MASGN (id: 1, line: 1, location: (1,0)-(1,17))*
#    +- nd_value:
#    |  @ NODE_RESCUE (id: 7, line: 1, location: (1,5)-(1,17))
#    |  +- nd_head:
#    |  |  @ NODE_VCALL (id: 2, line: 1, location: (1,5)-(1,6))
#    |  |  +- nd_mid: :p
#    |  +- nd_resq:
#    |  |  @ NODE_RESBODY (id: 6, line: 1, location: (1,7)-(1,17))
#    |  |  +- nd_args:
#    |  |  |  (null node)
#    |  |  +- nd_body:
#    |  |  |  @ NODE_FCALL (id: 4, line: 1, location: (1,14)-(1,17))
#    |  |  |  +- nd_mid: :p
#    |  |  |  +- nd_args:
:
```

生成規則を辿りながら必要に応じて各定義を見ていく

parse.yがやろうとしていることは抽象構文木(AST)を組み立てることなので、生成規則を辿りながら必要に駆られた時に各定義を参照していくのが最も歩きやすい。

(間違っても上から順に読んではいけない)

葉ノードから根に向かって辿ると常に親となる一つの非終端記号を辿ることになるので辿りやすい。※根から葉だとRHSは複数ある。

ここまでのまとめ

parse.yの歩き方

- parse.yは大まかにヘッダ部、型定義、生成規則、ユーザー定義部に分かれている
- parse.yはシンタックスハイライトがあると歩きやすい
- 生成規則を辿りながら必要に駆られた時に各定義を参照していくのが最も歩きやすい

04

「新しい」記法について

"深海に生きる魚族のように、自らが燃えなければ何処にも光はない"

――明石海人「白描」

Named References

構文規則の記号へのアクセスの際に
名前をつけて参照できる機能

アクション内で記号の値にアクセスする場合、`$1`や`$2`のようにインデックスを指定してアクセスすると位置がずれたり、可読性が少し悪い。`$name`のように終端・非終端記号の名前を用いて記号の値へアクセスすることができる。

複雑な構文解析を行う際に便利な機能であり、コードの見通しを良くし、保守性を高める役割を果たす。

Named References

以下の通り、置き換えが可能である

```
expression:
    term1 '+' term2 {
        $$ = $1 + $3;
    }
    | term1 '-' term2 {
        $$ = $1 - $3;
    };
```

```
expression:
    term1 '+' term2 {
        $$ = $term1 + $term2;
    }
    | term1 '-' term2 {
        $$ = $term1 - $term2;
    };
```

Named References

ただし、曖昧さが生じるため

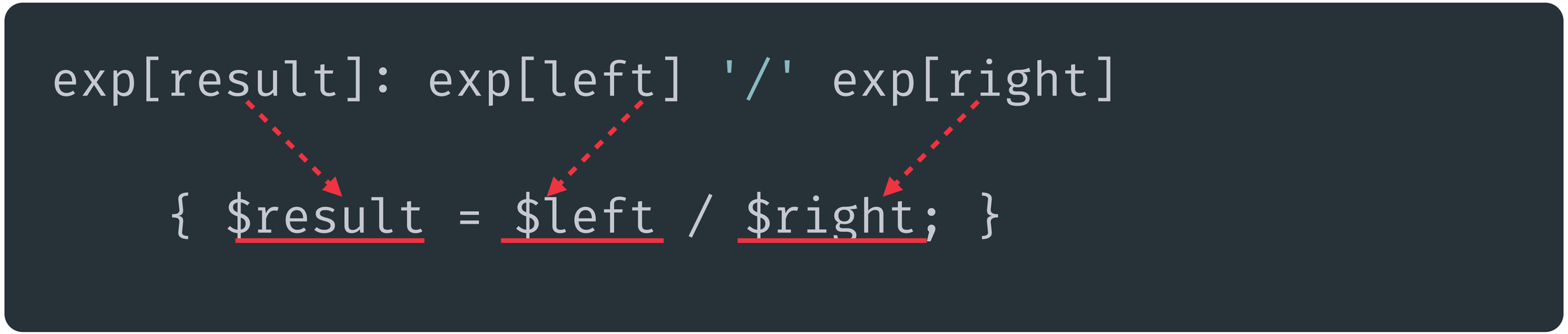
記号名だけでは判別ができないことがある

```
exp: exp '/' exp
  { $exp = $exp / $exp; } // $exp is ambiguous.

exp: exp '/' exp
  { $$ = $1 / $exp; } // One usage is ambiguous.

exp: exp '/' exp
  { $$ = $1 / $3; } // No error.
```

Named References

曖昧さを回避するために

各記号にエイリアスを設定可能



Parameterizing Rules

文法ファイルの生成規則の共通的なパターンのための シンタックスシュガーを提供する機能

生成規則の非終端記号の定義を、他の終端もしくは非終端記号でパラメータ化することができる機能。

OCaml向けのLR(1)パーサージェネレーターであるMenhirのアイデアを元に実装。

Menhir Reference Manual (version 20240715)

https://gallium.inria.fr/~fpottier/menhir/manual.html#sec32

Parameterizing Rules

以下のような頻出のパターンの共通化が可能

```
opt_args_tail:
    ',' args_tail {
        $$ = $2;
    }
    | /* none */ {
        $$ = new_args_tail( ... );
    };
```

```
opt_block_args_tail :
    ',' block_args_tail {
        $$ = $2;
    }
    | /* none */{
        $$ = new_args_tail( ... );
    }
;
```

Parameterizing Rules

異なる箇所は一つの非終端記号のみ

```
opt_args_tail:
    ',' args_tail {
        $$ = $2;
    }
    | /* none */ {
        $$ = new_args_tail( ... );
    };
```

```
opt_block_args_tail :
    ',' block_args_tail {
        $$ = $2;
    }
    | /* none */{
        $$ = new_args_tail( ... );
    }
;
```

Parameterizing Rules

Ruleの定義は以下の通りおこなう

```
%rule opt_args_tail(tail) <node_args>
                : ',' tail
                   {
                       $$ = $2;
                   }
                | /* none */
                   {
                       $$ = new_args_tail( ... );
                   }
                ;
```

ルール名

パラメータ

戻り値($$)の型

Parameterizing Rules

Ruleを使用する場合には以下の通り書き換えられる

```
opt_args_tail:
    ',' args_tail {
        $$ = $2;
    }
    | /* none */ {
        $$ = new_args_tail( ... );
    };
```

```
opt_args_tail:
    opt_args_tail(args_tail)
    ;
```

```
opt_block_args_tail :
    ',' block_args_tail {
        $$ = $2;
    }
    | /* none */{
        $$ = new_args_tail( ... );
    }
;
```

```
opt_block_args_tail:
    opt_args_tail(block_args_tail)
     ;
```

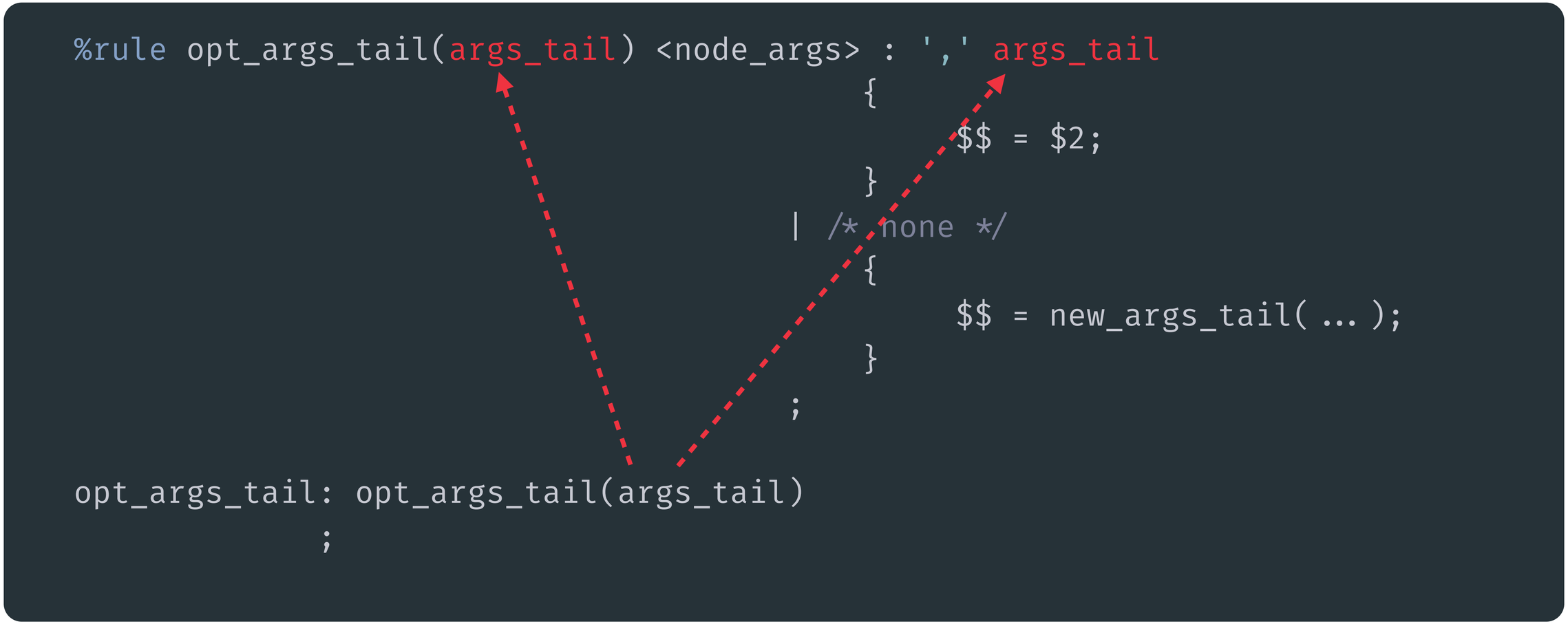
Parameterizing Rules

以下の通り展開する

```
%rule opt_args_tail(tail) <node_args> : ',' tail
                                          {
                                              $$ = $2;
                                          }
                                      | /* none */
                                          {
                                              $$ = new_args_tail( ... );
                                          }
                                      ;

opt_args_tail: opt_args_tail(args_tail)
             ;
```

Parameterizing Rules

以下の通り展開する



Standard library

汎用的な構造を持ったParameterizing Rulesのコレクションを提供する

一般的な構造を持つルールを標準ライブラリとして提供。

lib/lrama/grammar/stdlib.y に定義されており、構文解析の前に文法ファイル内に埋め込まれる。%no-stdlib ディレクティブが文法ファイルにあれば埋め込まれない。

ruby/lrama - lib/lrama/grammar/stdlib.y

https://github.com/ruby/lrama/blob/master/lib/lrama/grammar/stdlib.y

Standard library

次の3つのルールについては、
正規表現のような構文のエイリアスの使用が可能

名前	展開するルール	エイリアス
option(X)	ϵ | X	X?
list(X)	a possibly empty sequence of X’s	X*
nonempty_list(X)	a nonempty sequence of X’s	X+

Standard library

以下の通り、置き換えが可能である

```
opt_nl
     : /* empty */
     | '\n'
     ;
```

```
opt_nl
     : '\n'?
     ;
```

ここまでのまとめ

進化し続けるparse.y

- Named Referencesは構文規則の記号へのアクセスの際に位置情報($1)ではなく、名前を使用して参照できる機能
- Parameterizing Rulesは生成規則の共通パターンを抽出するための機能
- Standard Libraryが提供されており、?*+のようなエイリアスが提供されている

さいごに

parse.yは「たのしい」

- 現在の3.4-devではデフォルトパーサーがLramaがparse.yから生成するパーサーからPrismに変わったが、parse.yは今も進化を続けている
- 悪魔城や魔境や地獄といった印象も少しは変わったと思う
- LR構文解析器の力を信じているので、開発はこれからも続けていくつもり。今よりもっと たのしいparse.y を目指して

Stargaze at

github.com/ruby/lrama