Parser

Towards implementation of YAML parser

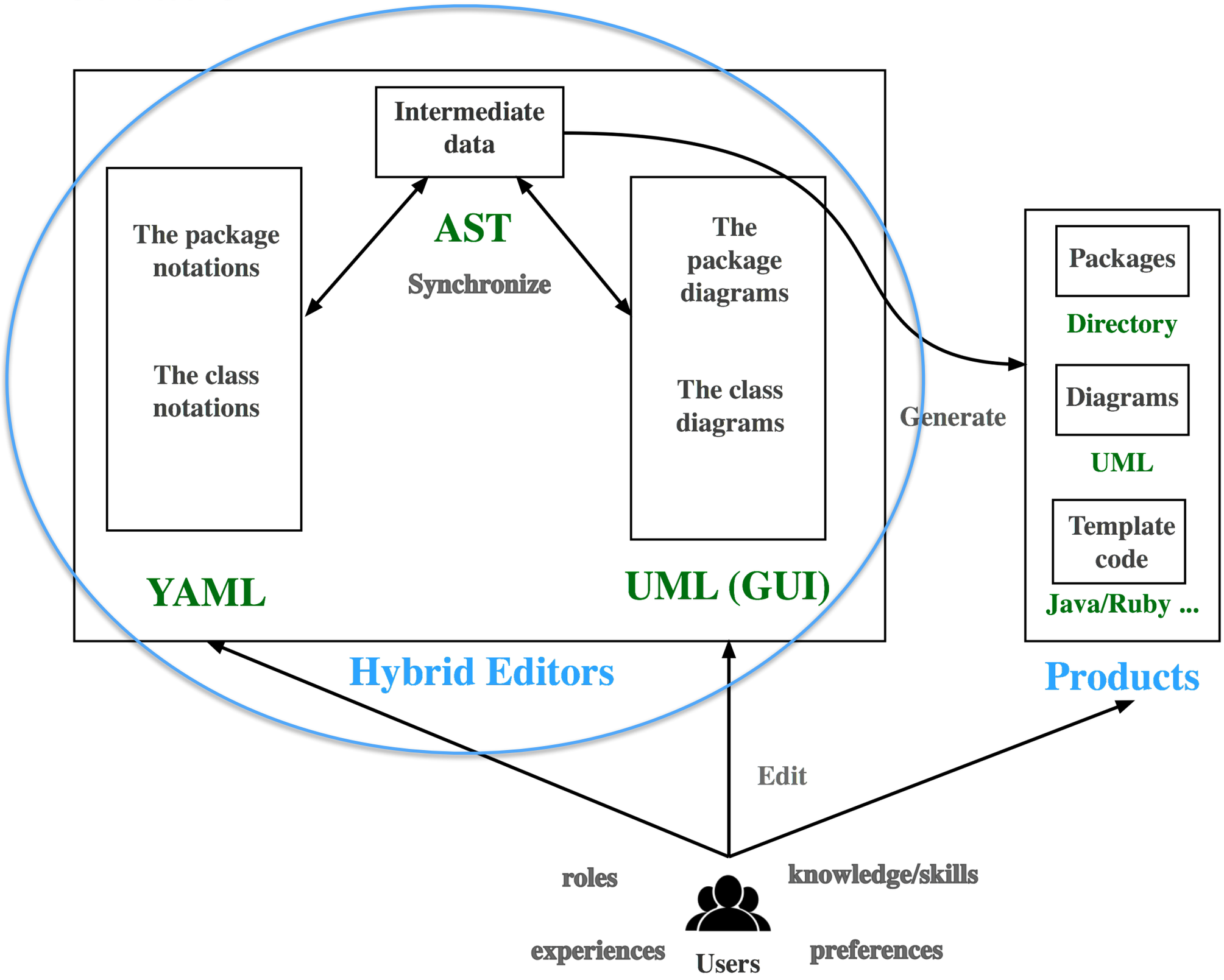
DBLab 141-F

d8161105 Mirai Watanabe

Outline

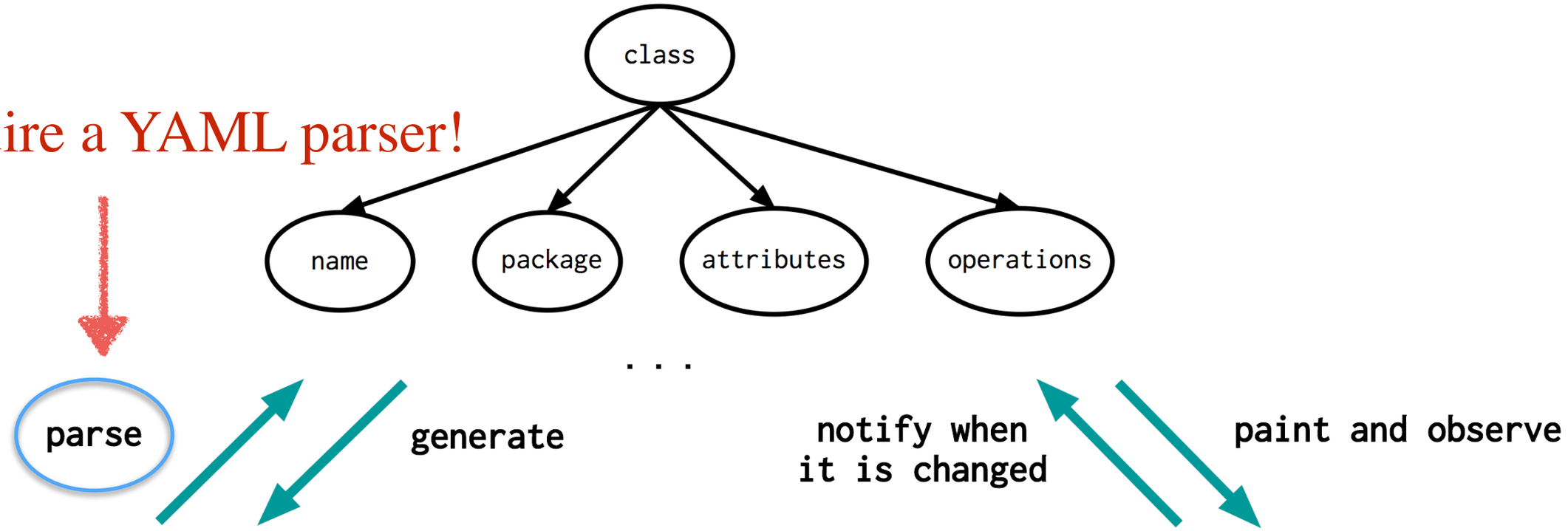
- Motivation
- Existing YAML parsers
- Parser
- Parser generator
- Combinator Parsing
- Example parsers

Motivation


Intermediate data
AST
Synchronize
The package notations
The class notations
YAML
The package diagrams
The class diagrams
UML (GUI)
Hybrid Editors
Generate
Packages
Directory
Diagrams
UML
Template code
Java/Ruby ...
Products
Edit
roles
knowledge/skills
experiences
Users
preferences

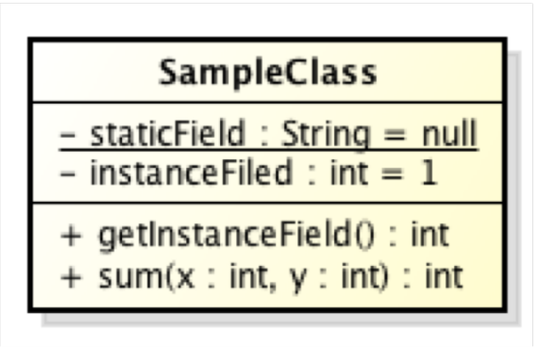
Motivation

We require a YAML parser!




```yaml
class:
  name: SampleClass
  package:  jp.ac.u-aizu
  attributes:
    - instance:
        visibility: '-'
        name: x
        type: int
        default: 0
  operations:
    - instance:
        {visibility: '+',
          name: getX,
          type: int}
```

textual syntax



visual syntax

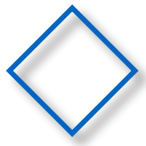
Existing YAML parsers

```
YAML Resources:
  YAML 1.2 (3rd Edition): http://yaml.org/spec/1.2/spec.html
  YAML 1.1 (2nd Edition): http://yaml.org/spec/1.1/
  YAML 1.0 (1st Edition): http://yaml.org/spec/1.0/

Projects:
  C/C++ Libraries:
  - libyaml        # "C" Fast YAML 1.1
  - Syck           # (dated) "C" YAML 1.0
  - yaml-cpp       # C++ YAML 1.2 implementation
  Ruby:
  - psych          # libyaml wrapper (in Ruby core for 1.9.2)
  - RbYaml         # YAML 1.1 (PyYaml Port)
  - yaml4r         # YAML 1.0, standard library syck binding
  Python:
  - PyYaml         # YAML 1.1, pure python and libyaml binding
  - PySyck         # YAML 1.0, syck binding
  Java:
  - JvYaml         # Java port of RbYaml
  - SnakeYAML      # Java 5 / YAML 1.1
  - YamlBeans      # To/from JavaBeans
  - JYaml          # Original Java Implementation
  Perl Modules:
  - YAML           # Pure Perl YAML Module
  - YAML::XS       # Binding to libyaml
  - YAML::Syck     # Binding to libsyck
  - YAML::Tiny     # A small YAML subset module
  - PlYaml         # Perl port of PyYaml
  C#/.NET:
  - yaml-net       # YAML 1.1 library
  - yatools.net    # (in-progress) YAML 1.1 implementation
```

Page not found…
Java 5 !?
Notice: I am no longer maintaining JYaml.

Scala library is not found…



Parser | How to create parser

- Create a **parser** (and lexical analyser).
  - It is difficult for even expert to create parser.

- Use a **parser generator**.
  - Domain specific languages (DSLs) are used.
  - Since error statements are complex, debug is difficult.

- Use a **Combinator Parsing**.
  - Internal DSLs are used.
  - Almost parser combinators seem context-free grammar.

Parser | Create parser

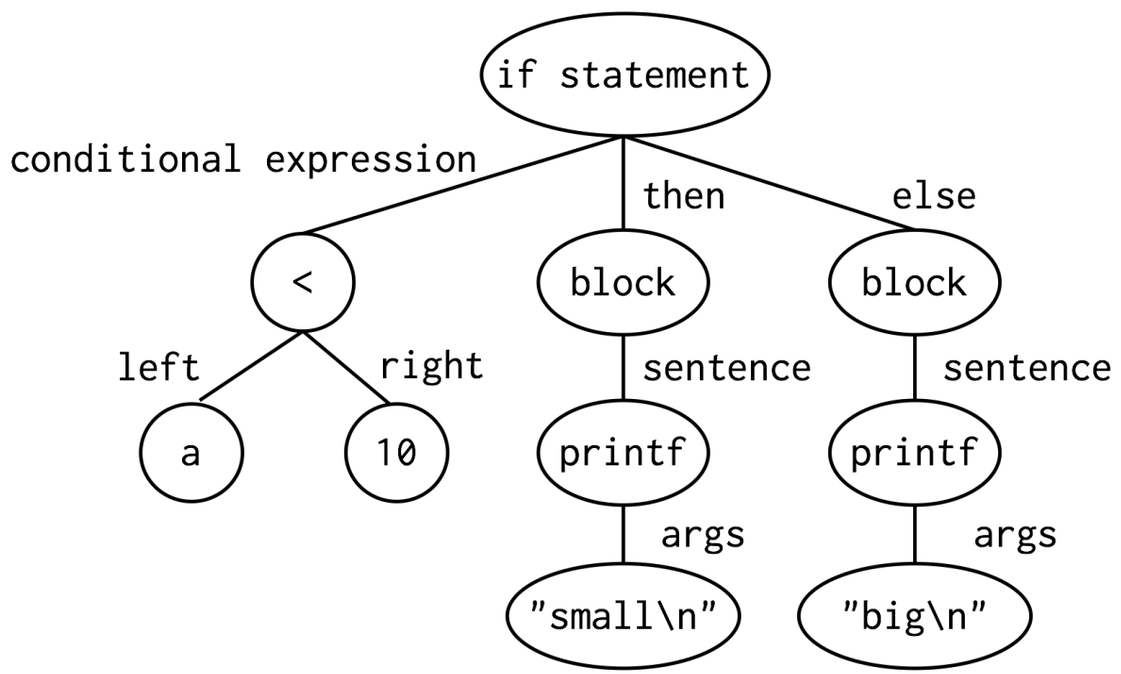
Target source of parsing

```
if (a < 10) {
    printf("small\n");
} else {
   printf("big\n");
}
```

Lexical analysis (scanner)

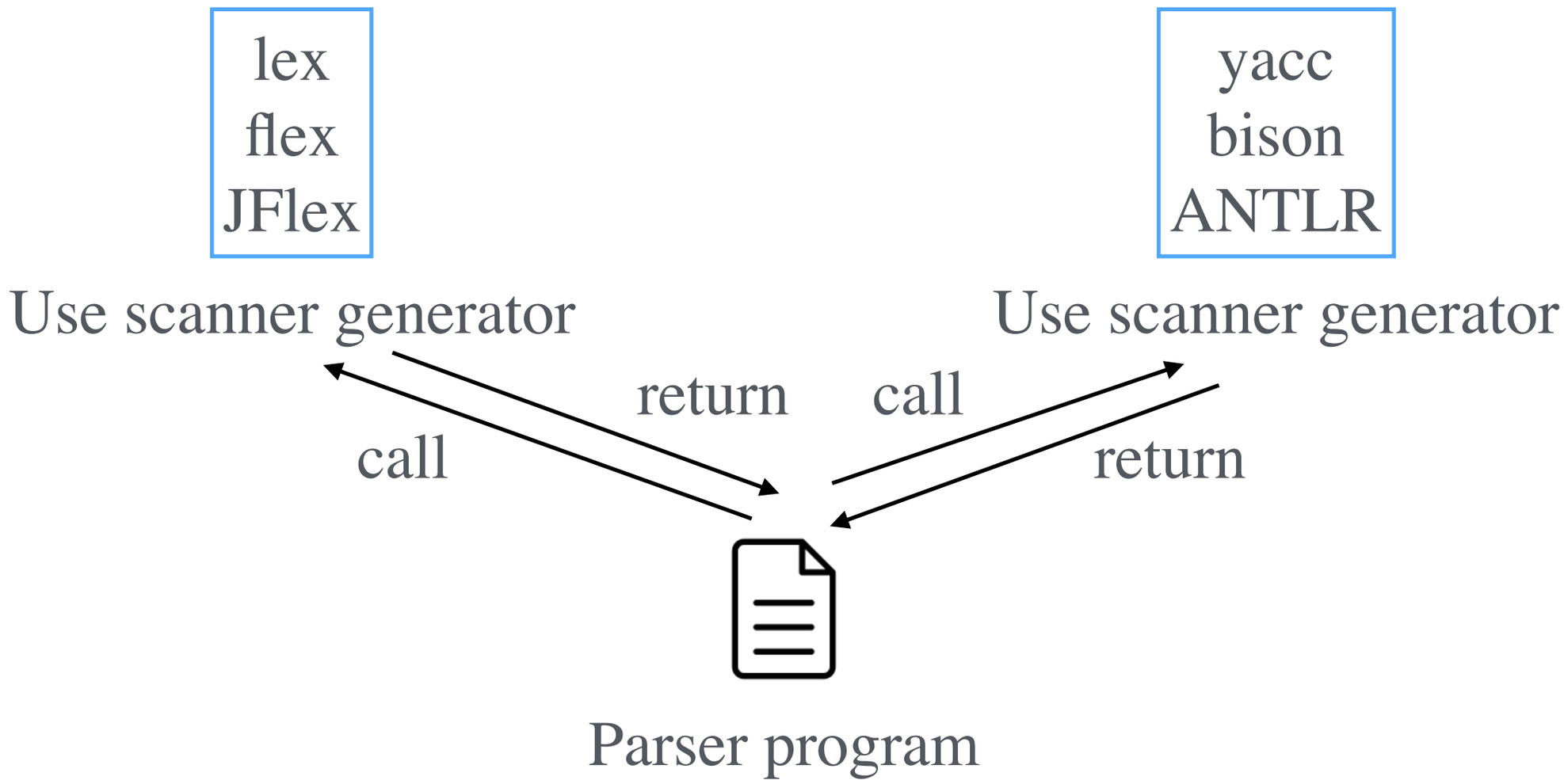
`if` `(` `a` `<` `10` `)` `{` `printf` `(` `“small\n”` `)` `;` `}` `else` `{` `printf` `(` `“big\n”` `)` `;` `}`

Parse



Parser | Use a parser generator

```
if (a < 10) {
    printf("small\n");
} else {
   printf("big\n");
}
```



Parser | lex and yacc

```
%{
#include <stdio.h>
#include "y.tab.h"

  int
    yywrap(void)
    {
      return 1;
    }
  %}
  %%
  "+"   return ADD;
  "-"   return SUB;
  "*"   return MUL;
  "/"   return DIV;
  "\n"  return CR;
  [1-9][0-9]* {
    double temp;
    sscanf(yytext,  "%lf",  &temp);
    yylval.double_value = temp;
    return DOUBLE_LITERAL;
  }
[0-9]*\.[0-9]* {
  double temp;
  sscanf(yytext,  "%lf",  &temp);
  yylval.double_value = temp;
  return DOUBLE_LITERAL;
}
%%
```

```
%{
#include <stdio.h>
  3:  #include <stdlib.h>
  4:  #define YYDEBUG 1
  5:  %}
%union {
    int           int_value;
    double        double_value;
}
 %token <double_value>      DOUBLE_LITERAL
 %token ADD SUB MUL DIV CR
 %type <double_value> expression term primary_expression
 %%
 line_list
     : line
     | line_list line
     ;
line
    : expression CR
    {
        printf(">>%lf\n",  $1);
    }
expression
    : term
    | expression ADD term
    {
        $$ = $1 + $3;
    }
    | expression SUB term
    {
        $$ = $1 - $3;
    }
    ;
    | term DIV primary_expression
    {
        $$ = $1 / $3;
    }
```

Combinator Parsing | Parser combinator

In functional programming, a **parser combinator** is a higher-order function that accepts several parsers as input and returns a new parser as its output. In this context, a parser is a function accepting strings as input and returning some structure as output, typically a parse tree or a set of indices representing locations in the string where parsing stopped successfully. Parser combinators enable a recursive descent parsing strategy that facilitates modular piecewise construction and testing. This parsing technique is called **combinatory parsing**.

Source: Parser combinator - wikipedia

Combinator Parsing | libraries

- Javascript - Parsimmon https://github.com/jneen/parsimmon
- Haskell - Parsec https://wiki.haskell.org/Parsec
- Ruby - rparsec http://docs.codehaus.org/display/JPARSEC/Ruby+Parsec
  - treetop https://github.com/nathansobo/treetop
- Python - parsy https://github.com/jneen/parsy
- Scala - Scala Parser Combinators https://github.com/scala/scala-parser-combinators

Parser combinator | Arithmetic Expression Parser

```
expr   ::= term { “+”  term ¦  “- “ term}.
term   ::= factor{ “*”  factor ¦  “/ “ factor}.
factor ::= floatingPointNumber ¦  “( “ expr  “)” .
```

```
{} : repeat
|  : selection
```

Backus-Naur form (BNF)

```
import scala.util.parsing.combinator._
class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep( “+” ~term ¦ “-” ~term)
  def term: Parser[Any] = factor~rep( “*” ~factor ¦ “/ “~factor)
  def factor: Parser[Any] = floatingPointNumber ¦ “( “~expr~” )”
}
```

Parser combinator in Scala

Parser combinator | Summary of parser combinators

""	: literal
"".r	: regular expression
P ~ Q	: sequential composition
P <~ Q	: sequential composition; keep left only
P ~> Q	: sequential composition; keep right only
P | Q	: alternative
opt(P)	: option
rep(P), P*	: repetition
repsep(P, Q)	: interleaved repetition
P ^^ f	: result conversion

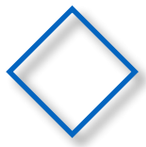
result values of parser

```
{
    "address book" : {
        "name" : "John Smith",
        "address" : {
                "street" : "10 market street",
                "city"   : "San Fransisco, CA",
                "zip"    : 94111
            },
        "phone number" : [
        "080-xxx-xxxx",
            "080-xxx-yyyy"
        ]
    }
}
```

```
(({~List((("address book” ~:)~(({~List((("name"~:)~
"John Smith"), (( “address"~:)~(({~List((("street"~:)~
"10 market street"), (("city"~:)~"San Fransisco, CA"),
(("zip"~:)~94111)))~})), (("phone number” ~:)~
(([~List("080-xxx-xxxx",  “080-xxx-yyyy"))
~])))~}))))~})
```

result values of parser | return rule

Parser combinator	Return type	Return value example	toString
`""`	String	“hoge”	hoge
`""".r`	String	“abc”	abc
`P ~ Q`	~(RP, RQ)	~(“true”,”?”)	true~?
`P | Q`	RP or RQ	“foo” …	foo …
`opt(P)`	Option[RP]	Some(“hoge”), None	“
`rep(P), P*`	List(RP)	List(“hoge”, “hoge”…)	“



Name parser

```
import util.parsing.combinator._
object NameParser extends RegexParsers {
  def firstName = "[a-zA-Z]+".r
  def lastName = "[a-zA-Z]+".r
  def fullName = firstName ~ lastName
  def parse(input: String) = parseAll(fullName, input)
}
```

Name parser | parser and spec

```scala
case class Name(firstName: String, middleName: Option[String], lastName: String)

object NameParser extends RegexParsers {
  def name: Parser[String] = "[a-zA-Z]+".r

  def fullName: Parser[Name] = rep(name) ^^ {
    /* リストの長さが4以上のものは値を捨てつつミドルネームアリ判定 */
    case names@List(first, middle, last, _*) => Name(first, Some(middle), last)
    case List(first, last) => Name(first, None, last)
  }

  def parse(input: String) = parseAll(fullName, input)
}
```

```scala
class ParserSpec extends FlatSpec with Matchers {
  "Martin Odersky" should "has FirstName and Last name" in {
    val parseResult = NameParser.parse("Martin Odersky")
    val name = parseResult.get
    name should be(Name("Martin", None, "Odersky"))
  }

 "John F Kennedy" should "has FirstName, Middle name and Last name" in {
    val parseResult = NameParser.parse("John F Kennedy")
    val name = parseResult.get
    name should be(Name("John", Some("F"), "Kennedy"))
  }

}
```

CSV parser | parser

```
abstract class Row

case class HeaderRow(cells: List[String]) extends Row {
  override def toString: String = {
    s"Header: $cells"
  }
}

case class DataRow(cells: List[String]) extends Row {
  override def toString: String = {
    s"Data: $cells"
  }
}
```

```
object CsvParser extends RegexParsers {
  def eol = opt('\r') <~ '\n'

  def line = ".*".r <~ eol

  def headerRow = line ^^ { row => new HeaderRow(row.split(",").toList) }

  def dataRow = line ^^ { row => new DataRow(row.split(",").toList) }

  def all = headerRow ~ rep(dataRow) ^^ { res => res._1 :: res._2 }

  def parse(input: String): ParseResult[List[Row]] = parseAll(all, input)

}
```

CSV parser | spec

```
class ParserSpec extends FlatSpec with Matchers{
  "CSV parser" should "return Row List" in {
    val parseResult = CsvParser.parse(
      """name,age,place
        John,17,NewYork
        Mike,23,Soul
      """)
    val lines = parseResult.get
    lines should be(
      List(HeaderRow(List("name", "age", "place")), DataRow(List("John", "17", "NewYork")), DataRow(List("Mike", "23", "Soul")
  }

}
```

Json parser | parser

```
object JsonParser extends RegexParsers {

  def stringLiteral: Parser[String] = "\"[-a-zA-Z0-9:*/+,#$%& ]+\"".r ^^ {
    _.replaceAll("\"", "")
  }

  def intLiteral: Parser[Int] = """[1-9][0-9]*|0""".r ^^ {
    _.toInt
  }

  def floatingPointNumber: Parser[Double] = """-?[0-9]+\.[0-9]+""".r ^^ {
    _.toDouble
  }

  def value: Parser[Any] = obj | arr |
    stringLiteral | floatingPointNumber | intLiteral |
    "null" ^^ { _ => null } | "true" ^^ { _ => true } | "false" ^^ { _ => false }

  def obj: Parser[Map[String, Any]] = "{" ~> repsep(member, ",") <~ "}" ^^ {
    Map() ++ _
  }

  def arr: Parser[List[Any]] = "[" ~> repsep(value, ",") <~ "]"

  def member: Parser[(String, Any)] = stringLiteral ~ ":" ~ value ^^ { case k ~ ":" ~ v => (k, v) }

  def parse(input: String) = parseAll(value, input)
}
```

Json parser | spec

```
class ParserSpec extends FlatSpec with Matchers {
  "Json parser" should "return lisp like value" in {

    val parseResult = JsonParser.parse( """
    {
      "address book" : {
          "name" : "John Smith",
          "address" : {
            "street" : "10 market street",
            "city"   : "San Fransisco, CA",
            "zip"    : 94111
          },
          "phone number" : [
            "080-xxx-xxxx",
            "080-xxx-yyyy"
          ]
      }
    }""")

    parseResult.get should be === Map("address book" ->
      Map(
        "name" -> "John Smith",
        "address" -> Map(
          "street" -> "10 market street", "city" -> "San Fransisco, CA", "zip" -> 94111
        ),
        "phone number" -> List("080-xxx-xxxx", "080-xxx-yyyy")
      )
    )
  }
}
```

Arithmetic interpreter I parser

```
trait AST

case class AddOp(left: AST, right: AST) extends AST

case class SubOp(left: AST, right: AST) extends AST

case class MulOp(left: AST, right: AST) extends AST

case class IntVal(value: Int) extends AST


object ArithExprParser extends RegexParsers {

  def intLiteral: Parser[AST] = """[1-9][0-9]*|0""".r ^^ {
    case value => IntVal(value.toInt)
  }

  def expr: Parser[AST] = chainl1(term, calc("+") | calc("-"))

  def calc(operand: String) = operand ^^ { op => (left: AST, right: AST) =>
    op match {
      case "+" => AddOp(left, right)
      case "-" => SubOp(left, right)
    }
  }

  def term: Parser[AST] = chainl1(factor, "*" ^^ { op => (l: AST, r: AST) => MulOp(l, r) })

  def factor: Parser[AST] = intLiteral | "(" ~> expr <~ ")"

  def parse(input: String): ParseResult[AST] = parseAll(expr, input)
}
```

Arithmetic interpreter | visitor pattern in Scala

```
object ArithExprEvaluator {
  def eval(ast: AST): Any = ast match {
    case AddOp(left, right) =>
      (eval(left), eval(right)) match {
        case (lval: Int, rval: Int) => lval + rval
      }
    case SubOp(left, right) =>
      (eval(left), eval(right)) match {
        case (lval: Int, rval: Int) => lval - rval
      }
    case MulOp(left, right) =>
      (eval(left), eval(right)) match {
        case (lval: Int, rval: Int) => lval * rval
      }
    case IntVal(value) => value
  }

}
```

Arithmetic interpreter | spec

```
class ParserSpec extends FlatSpec with Matchers {
  "ArithExpr parser" should "return AST" in {
    val parseResult = ArithExprParser.parse("((4 + 2) * 3) - 6")
    parseResult.get should be(SubOp(MulOp(AddOp(IntVal(4), IntVal(2)), IntVal(3)), IntVal(6)))
  }

  "ArithExpr evaluator" should "return value" in {
    val parseResult = ArithExprParser.parse("((4 + 2) * 3) - 6")
    ArithExprEvaluator.eval(parseResult.get) should be(12)
  }


}
```

Reference