The Little Leex & Yecc

ZhenyuanLua

前言

Leex & Yecc 是 Erlang 版的 Flex & Yacc(Bison).

Flex & Yacc 用于开发编译器和解释器, 适用于任何使用模式匹配输入的应用,

如计算器/解释器/编译器/领域特定语言.

Flex & Yacc 应用领域, Leex & Yecc 亦可往之.

本书内容

较 <Flex & Bison>, 本书先理解 Leex & Yecc 规范, 使用 Leex & Yecc, 增加语法可视化内容.

具体内容如下:

第一章, 简介语法分析相关知识.

第二章, 简介 BNF 文法, W3C EBNF 记法, 语法可视化.

第三章, 理解 Leex 规范, 熟悉 Leex 词法文件定义, 生成的词法解析器代码.

第四章, 理解 Yecc 规范, 熟悉 Yecc 语法文件定义. 生成的语法解析器代码.

第五章, Leex 实战, 使用 Leex 开发单词统计应用.

第六章, Yecc 实战, 使用 Yecc 联合 Leex 开发计算器应用.

第七章, 基于 Yecc/Leex 实现 JSON 解释器.

第八章, 理解 LALR-1 分析算法.

排版约定

本书应用了以下排版约定:

标点符号: 使用英文标点加空格.

本书范例

本书的范例程序可以在线获得:

```
https://www.github.com/zhenyuanlau/leex-yecc-book/code
```

致谢

- Flex & Bison
- TheBeamBook
- AsciiDoc

Chapter 1. 简介

代码分析过程分为词法分析和语法分析两个部分:

- 词法分析, 把文本输入转换成 Token 流.
- 语法分析, 把 Token 流转换成 AST.

理解 Elixir 编译过程， 词法分析和语法分析是绕不过的.

1.1. 词法分析

Leex 生成的词法分析器, 读取输入, 匹配输入与指定的正则表达式, 执行匹配所关联的 Erlang 代码.
词法分析是在输入中寻找字符的模式, 正则表达式是对模式描述的工具.

1.1.1. 词法分析器

词法分析器可以由 Leex 生成.

1.1.2. 正则表达式

1.2. 语法分析

语法由一系列规则组成, 语法分析器基于语法规则识别语法上正确的输入.
语法分析器不保证语义正确.
语法分析器的任务把 Token 流转换成 AST.

1.2.1. BNF 文法

BNF 是上下文无关文法的标准, 用来描述 Token 流转换成 AST 的规则.

1.2.2. 语法分析器

语法分析器可以由 Yecc 生成.

1.3. 手写词法分析器

Chapter 2. BNF 文法

2.1. 语法描述

W3C EBNF Syntax

```
Grammar  ::= Production*
Production
         ::= NCName '::=' ( Choice | Link )
NCName   ::= [http://www.w3.org/TR/xml-names/#NT-NCName]
Choice   ::= SequenceOrDifference ( '|' SequenceOrDifference )*
SequenceOrDifference
         ::= (Item ( '-' Item | Item* ))?
Item     ::= Primary ( '?' | '*' | '+' )*
Primary  ::= NCName | StringLiteral | CharCode | CharClass | '(' Choice
')'
StringLiteral
         ::= '"' [^"]* '"' | "'" [^']* "'"
CharCode ::= '#x' [0-9a-fA-F]+
CharClass
         ::= '[' '^'? ( Char | CharCode | CharRange | CharCodeRange )+
']'
Char     ::= [http://www.w3.org/TR/xml#NT-Char]
CharRange
         ::= Char '-' ( Char - ']' )
CharCodeRange
         ::= CharCode '-' CharCode
Link     ::= '[' URL ']'
URL      ::= [^#x5D:/?#]+ '://' [^#x5D#]+ ('#' NCName)?
Whitespace
         ::= S | Comment
S        ::= #x9 | #xA | #xD | #x20
Comment  ::= '/*' ( [^*] | '*'+ [^*/] )* '*'* '*/'
```

2.2. EBNF 相关概念

2.2.1. 语法

语法是产生规则的集合.

2.2.2. 产生规则

产生规则格式如下:

```
symbol ::= expression
```

::= 左边的语法符号称为规则的左手边(LHS, left-hand side),

::= 右边的语法符号称为规则的右手边(RHS, right-hand side).

2.2.3. 终结符

终结符只能出现在规则的右手边, 可以是符号/字符串/模式.

2.2.4. 非终结符

非终结符可以出现在规则的左手边, 也可以出现在规则的右手边.

2.2.5. 语法操作符

Table 1. 语法操作符表

操作符	说明
A?	可选 A
A B	匹配 A 后跟着 B
A | B	匹配 A 或者 B
A - B	匹配 A 不匹配 B
A+	至少匹配一个 A
A*	匹配 0 个或多个 A

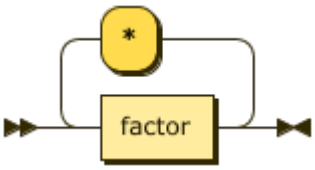
2.2.6. EBNF 代码示例

算术表达式语法

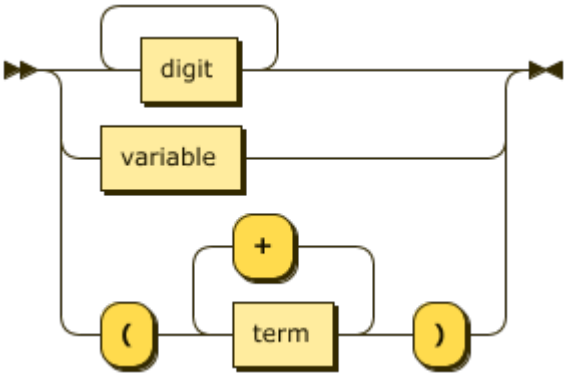
```
expression ::= term | term "+" expression
term       ::= factor | factor "*" term
factor     ::= constant | variable | "(" expression ")"
variable   ::= "x" | "y" | "z"
constant   ::= digit | digit constant
digit      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

2.3. 语法可视化

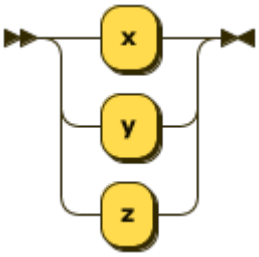
```
term ::= factor | factor "*" term
```



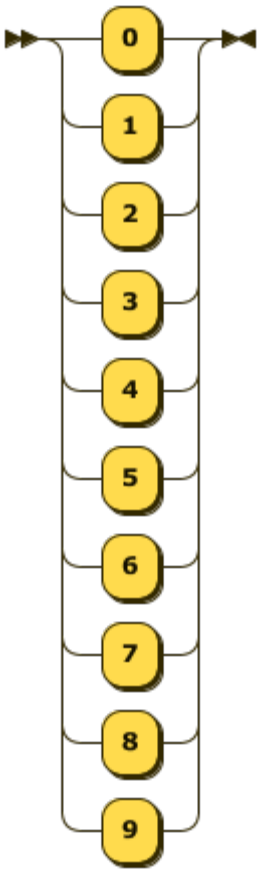

```
factor ::= constant | variable | "(" expression ")"
```




```
variable ::= "x" | "y" | "z"
```




```
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```


0
1
2
3
4
5
6
7
8
9

Chapter 3. Leex 参考

Leex 是 Erlang 的一个模块，基于正则表达式生成词法解析器.

3.1. 词法解析器导出函数

关注 Module:string(String) 函数.

```
Module:string(String) -> StringRet
```

tokenizer.erl

```
-spec string(String :: string()) -> {ok, [Token], EndLine} | ErrorInfo
```

3.2. 词法文件格式

tokenizer.xrl.template

```
% <Header>

Definitions.

% <Macro Definitions>

Rules.

% <Token Rules>

Erlang code.

% <Erlang code>
```

3.2.1. 结构要求

Definitions/Rules/Erlang code 必须要有.

<Header>/<Macro Definitions>/<Erlang Code> 是可选的.

Token Rules 至少要有一条.

3.2.2. 格式要求

Macro Definition 格式

```
NAME = VALUE
```

Token Rule 格式

忽略 PushBackList 相关内容.

```
<Regexp> : <Erlang code>.
```

如果 <Regexp> 匹配成功， 则求值关联的 <Erlang code>.

在 <Erlang code> 中，可以使用变量:

- TokenChars
- TokenLen
- TokenLine

<Erlang code> 返回值:

- {token, Token}
- {end_token, Token}
- skip_token
- {error, ErrString}

3.3. 正则表达式

在这里，合法的正则表达式是 egrep/awk 正则表达式的一个子集.

3.3.1. 正则表达式

Table 2. 正则表达式

表达式	说明
c	Matches the non-metacharacter c.
\c	Matches the escape sequence or literal character c.
.	Matches any character.
^	Matches the beginning of a string.
$	Matches the end of a string.

`[abc…]`	Character class, which matches any of the characters abc…. Character ranges are specified by a pair of characters separated by a –.
`[^abc…]`	Negated character class, which matches any character except abc….
`r1 | r2`	Alternation. It matches either r1 or r2.
`r1r2`	Concatenation. It matches r1 and then r2.
`r+`	Matches one or more rs.
`r*`	Matches zero or more rs.
`r?`	Matches zero or one rs.
`(r)`	Grouping. It matches r.

3.3.2. 转义序列

Table 3. 转义序列

字符序列	说明
`\b`	Backspace.
`\f`	Form feed.
`\n`	Newline (line feed).
`\r`	Carriage return.
`\t`	Tab.
`\e`	Escape.
`\v`	Vertical tab.
`\s`	Space.
`\d`	Delete.

`\ddd`	The octal value ddd.
`\xhh`	The hexadecimal value hh.
`\x{h…}`	The hexadecimal value h….
`\c`	Any other character literally, for example \\ for backslash, \" for ".

3.4. 重点

`Token` 是用户自定义的， 是一个 Erlang 元组.

正则表达式关联的 Erlang 代码, 可以使用变量 `TokenChars`/`TokenLen`/`TokenLine`.

正则表达式关联的 Erlang 代码，必须返回

- `{token, Token}`
- `{end_token, Token}`
- `skip_token`
- `{error, ErrString}`

生成的词法解析器导出函数 `Module:string(String) -> {ok, [Token], EndLine} | ErrorInfo`.

Chapter 4. Yecc 参考

`Yecc` 是 Erlang 的一个模块, 用于生成 LALR-1 语法分析器.

4.1. 语法分析器导出函数

关注 `Module:parse(Tokens)` 函数.

parser.erl

```
-type yecc_ret() :: {'error', _} | {'ok', _}.
-spec parse(Tokens :: list()) -> yecc_ret().
```

4.2. 语法文件格式

parser.yrl.template

```
% <Header>

% nonterminal categories

Nonterminals sentence nounphrase verbphrase.

% Terminal categories

Terminals article adjective noun verb.

% rootsymbol
Rootsymbol sentence.

% <Endsymbol>

% operator precedence & associative

% grammar rules

Erlang code.

% <Expect>
```

4.2.1. 结构要求

`Nonterminals`/`Terminals`/`Rootsymbol` 必须要有.

`<Grammer Rules>` 至少要有一条.

4.2.2. 格式要求

`Operator Precedence` 格式

```
associative precedence operator
```

`Grammer Rule` 格式

```
Left_hand_side -> Right_hand_side : Associated_code.
```

左手边是一个非终结符. 右手边是一个用空格分隔的序列, 序列元素是可以是终结符, 也可以是非终结符.

关联的代码是一个 Erlang 表达式序列.

如果 Erlang 表达式序列为空, 则语法规则的分隔符 `:` 也可以省略.

其他符号必须使用单引号引起来, 且不能与 `'$empty'`, `'$end'`, `'$undefined'` 等符号同名.

当语法规则匹配成功时, 语法分析器会求值 Erlang 表达式序列. 最后一个表达式的值就是语法符号的语义值.

关联的 Erlang 代码可以包含伪变量 `'$1'`, `'$2'`, `'$3'`..., 用来引用语法规则右手边符号序列元素的语义值.

如果伪变量绑定的符号是终结符, 那么伪变量的值就是一个 Token. Token 可以包含位置信息.

如果没有关联 Erlang 代码, 语法符号关联的就是 `'$undefined'`.

语法规则的右手边可以为空, 用特殊符号 `'$empty'` 表示.

4.3. BNF 文法

见 <BNF 文法>.

4.4. 重点

语法分析器导出函数 `Module:parse(Tokens)`.

理解伪变量 `'$1'`, `'$2'`, `'$3'`, ...

理解特殊变量 `'$empty'`, `'$undefined'`, `'$end'`.

理解 Yecc 工作原理, 见 <理解 LALR-1 分析算法>.

Chapter 5. Leex 实战

5.1. 单词统计应用

5.1.1. 词法文件

wc.xrl

```
Definitions.

Char = [a-zA-Z]
EOL = \n

Rules.

{Char}+ :
  {token, {word, TokenLen}}.
{EOL} :
  {token, {line, 1}}.
. :
  {token, {char, 1}}.

Erlang code.

-export([count/1]).

count(String) ->
  {ok, Tokens, _} = string(String),
  LineCount = lists:sum([element(2, T) || T <- Tokens, element(1, T) ==
line]),
  WordCount = lists:sum([1 || T <- Tokens, element(1, T) == word]),
  CharCount = lists:sum([element(2, T) || T <- Tokens]),
  io:format("Lines: ~p Words: ~p Chars: ~p~n", [LineCount, WordCount,
CharCount]).
```

5.1.2. 构建

Makefile

```
Q := @
```

```
WC := wc.erl

default: shell

$(WC): wc.xrl
    $(Q) mkdir -p ebin
    $(Q) erlc -o $@ +'{verbose,true}' +'{report,true}' $<
    $(Q) erlc -o ebin $(WC)

shell: clean $(WC)
    erl -pa ebin

clean:
    $(Q) rm -fr ebin $(WC)
```

5.1.3. 执行

```
$ make shell
> wc:count("hello, leex!\n").
```

Chapter 6. Yecc 实战

计算器应用.

6.1. 语法文件

calc_parser.yrl

```
Nonterminals expression term number.

Terminals integer float '+' '-' '*' '/'.

Rootsymbol expression.

expression -> expression '+' term : '$1' + '$3'.
expression -> expression '-' term : '$1' - '$3'.
expression -> term : '$1'.

term -> term '*' number : '$1' * '$3'.
term -> term '/' number : '$1' / '$3'.
term -> number : '$1'.

number -> integer : ?exprs('$1').
number -> float : ?exprs('$1').

Erlang code.

-export([eval/1]).

-define(id(Token), element(1, Token)).
-define(location(Token), element(2, Token)).
-define(exprs(Token), element(3, Token)).

eval(String) ->
    {ok, Tokens, _} = calc_tokenizer:string(String),
    io:format("Tokens: ~w~n", [Tokens]),
    case calc_parser:parse(Tokens) of
        {ok, Forms} ->
            io:format("Forms: ~w~n", [Forms]);
        {error, Error} ->
            io:format("Error: ~p~n", [Error])
```

```
    end.
```

6.2. 词法文件

calc_tokenizer.xrl

```
Definitions.

D = [0-9]
WhiteSpace = [\s\t]
OpCode = [\+\*\-/]

Rules.

{D}+ :
    {token,{integer, TokenLine, list_to_integer(TokenChars)}}.

{D}+\.{D}+((E|e)(\+|\-)?{D}+)? :
    {token,{float, TokenLine, list_to_float(TokenChars)}}.

{WhiteSpace} : skip_token.
{OpCode} :
    {token, {list_to_atom(TokenChars), TokenLine,
list_to_atom(TokenChars)}}.

Erlang code.
```

6.3. 构建

Makefile

```
Q := @

CALC_TOKENIZER := calc_tokenizer.erl
CALC_PARSER := calc_parser.erl

default: shell

calc: $(CALC_PARSER) $(CALC_TOKENIZER)

$(CALC_TOKENIZER): calc_tokenizer.xrl
    $(Q) mkdir -p ebin
```

```
    $(Q) erlc -o $@ +'{verbose,true}' +'{report,true}' $<
    $(Q) erlc -o ebin $(CALC_TOKENIZER)

$(CALC_PARSER): calc_parser.yrl
    $(Q) mkdir -p ebin
    $(Q) erlc -o $@ +'{verbose,true}' +'{report,true}' $<
    $(Q) erlc -o ebin $(CALC_PARSER)

shell: clean $(CALC_TOKENIZER) $(CALC_PARSER)
    $(Q) erl -pa ebin

clean:
    $(Q) rm -fr ebin $(CALC_PARSER) $(CALC_TOKENIZER)
```

6.4. 执行

```
$ make
> calc_parser:eval("1 + 1").
```

Chapter 7. JSON 解析器

7.1. 语法文件

json_parser.yrl

```
Nonterminals value values object array pair pairs.

Terminals number string true false null '[' ']' '{' '}' ',' ':'.

Rootsymbol value.

value -> object  :  '$1'.
value -> array   :  '$1'.
value -> number  :  get_val('$1').
value -> string  :  get_val('$1').
value -> 'true'  :  get_val('$1').
value -> 'null'  :  get_val('$1').
value -> 'false' :  get_val('$1').

object -> '{' '}' : #{}.
object -> '{' pairs '}' : '$2'.

pairs -> pair : '$1'.
pairs -> pair ',' pairs : maps:merge('$1', '$3').

pair -> string ':' value : #{ get_val('$1') => '$3' }.

array -> '[' ']' : {}.
array -> '[' values ']' : list_to_tuple('$2').

values -> value : [ '$1' ].
values -> value ',' values : [ '$1' | '$3' ].



Erlang code.

-export([demo/0]).

demo() ->
    {ok, Tokens, _} = json_tokenizer:t(),
```

```
    case json_parser:parse(Tokens) of
        {ok, Forms} ->
            io:format("Forms: ~w~n", [Forms]);
        {error, Error} ->
            io:format("Error: ~p~n", [Error])
    end.

get_val({_,_,Val}) -> Val;
get_val({Val, _}) -> Val.
```

7.2. 词法文件

json_tokenizer.xrl

```
Definitions.

Digit         = [0-9]
Digit1to9     = [1-9]
HexDigit      = [0-9a-f]
UnescapedChar = [^\"\\]
EscapedChar   = (\\\\)|(\\")|(\\b)|(\\f)|(\\n)|(\\r)|(\\t)|(\\/)
Unicode       = (\\u{HexDigit}{HexDigit}{HexDigit}{HexDigit})
Quote         = [\"]
Delim         = [\[\]:,{}]
Space         = [\n\s\t\r]

Rules.

{Quote}{Quote} : {token, {string, TokenLine, ""}}.
{Quote}({EscapedChar}|({UnescapedChar})|({Unicode}))+{Quote} :
  {token, {string, TokenLine, drop_quotes(TokenChars)}}.

null  : {token, {null,  TokenLine}}.
true  : {token, {true,  TokenLine}}.
false : {token, {false, TokenLine}}.

{Delim} : {token, {list_to_atom(TokenChars), TokenLine}}.

{Space} : skip_token.

-?{Digit1to9}+{Digit}*\.{Digit}+((E|e)(\+|\-)?{Digit}+)? :
  {token, {number, TokenLine, list_to_float(TokenChars)}}.
-?{Digit1to9}+{Digit}* :
```

```
  {token, {number, TokenLine, list_to_integer(TokenChars)+0.0}}.

Erlang code.
-export([t/0]).

drop_quotes([$" | QuotedString]) ->
literal(lists:droplast(QuotedString)).
literal([$\\,$" | Rest]) ->
  [$"|literal(Rest)];
literal([$\\,$\\ | Rest]) ->
  [$\\|literal(Rest)];
literal([$\\,$/ | Rest]) ->
  [$/|literal(Rest)];
literal([$\\,$b | Rest]) ->
  [$\b|literal(Rest)];
literal([$\\,$f | Rest]) ->
  [$\f|literal(Rest)];
literal([$\\,$n | Rest]) ->
  [$\n|literal(Rest)];
literal([$\\,$r | Rest]) ->
  [$\r|literal(Rest)];
literal([$\\,$t | Rest]) ->
  [$\t|literal(Rest)];
literal([$\\,$u,D0,D1,D2,D3|Rest]) ->
  Char = list_to_integer([D0,D1,D2,D3],16),
  [Char|literal(Rest)];
literal([C|Rest]) ->
  [C|literal(Rest)];
literal([]) ->[].

t() ->
  {ok,
   [{'{',1},
    {string,2,"no"},
    {':',2},
    {number,2,1.0},
    {'}',3}
   ],
   4}.
```

7.3. 構建

Makefile

```
Q := @

JSON_TOKENIZER := json_tokenizer.erl
JSON_PARSER := json_parser.erl

default: shell

json: $(JSON_PARSER) $(JSON_TOKENIZER)


$(JSON_TOKENIZER): json_tokenizer.xrl
    $(Q) mkdir -p ebin
    $(Q) erlc -o $@ +'{verbose,true}' +'{report,true}' $<
    $(Q) erlc -o ebin $(JSON_TOKENIZER)

$(JSON_PARSER): json_parser.yrl
    $(Q) mkdir -p ebin
    $(Q) erlc -o $@ +'{verbose,true}' +'{report,true}' $<
    $(Q) erlc -o ebin $(JSON_PARSER)


shell: clean $(JSON_TOKENIZER) $(JSON_PARSER)
    $(Q) erl -pa ebin

clean:
    $(Q) rm -fr ebin $(JSON_TOKENIZER) $(JSON_PARSER)
```

7.4. 執行

```
$ make
> json_parser:demo().
```

Chapter 8. 理解 LALR-1 分析算法

8.1. 移进

8.2. 规约

参考

W3C EBNF Notation

Railroad Diagram Generator

EDoc Leex

EDoc Yecc