LR(1) 语法分析*

– Based on Racket (A Dialect of Lisp)

61519322 杨哲睿

2021 年 11 月 20 日

目录

[*] 两次实验的报告和源码都可以在我的*gitee repo*中找到。

1 实验要求

1. 手工画出 LR(1) 项目集族及状态图
2. 构造 LR(1) 分析表
3. 在上述过程中，对二义文法进行必要的处理

   表达优先级和结合律

   按照优先级和结合律对预测符及状态转换关系进行必要的取舍
4. 按照 LR(1) 分析表写出语法分析程序

当然，这只是一个最简单的要求。但考虑到 LR(1) 分析器的手工构造难度，该实验使用**程序推导**的方式来构造 LR(1) 项目集族和状态转换图。并且在实验的具体实现上有所变化。

2 设计思路

根据 htdp 中的详细介绍，我们可以将设计一个程序分为以下六个步骤：

1. From Problem to Data Definitions
2. Signature, Purpose Statement, Header
3. Functional Examples
4. Function Template
5. Function Definitions
6. Testing

报告也将从这六个方面展开，对于该语法分析程序的制作进行分析。

2.1 From Problem to Data Definitions

Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

总的来说，我们 LR 分析程序：

- 输入为：Token 的流 (具体实现为生成器 *generator*)
- 输出为：对应的一颗语法树

一个 LR(1) 语法分析程序所涉及的数据结构如下：

1. `syntax-item` 文法符号 – 指上下文无关文法中的非终结符或终结符单元，拥有名称 `id`，优先级 `proority`、匹配器 (函数)`matcher`
2. `matched-item` 匹配的文法符号 – 指输出时匹配后的终结符单元
3. `production` 产生式 – 上下文无关文法的一个产生式，分为左部和右部两部分

4. `syntax-tree-node` 语法树的一个节点
5. `LRItem` LR(1) 项 – 分为四个部分：产生式的左部、产生式右部在 · 之前的部分、产生式右部在 · 之后的部分、展望 *(Look-Ahead)* 符号

对该数据结构的定义如下：

Listing 1: types.rkt

```
(struct syntax-item
 (id priority matcher) #:transparent)
(struct matched-item
 (stx content))
(struct production
 (left right) #:transparent)
(struct syntax-tree-node
 (head children) #:transparent)
```

当然这些只是基础的定义，为了完成 LR(1) 程序，我们还需要定义：

1. 一个 LR(1) 项目集就是一个 `LRItem` 的列表
2. 一个上下文无关文法包含：终结符号、非终结符号、开始符号、产生式
3. 对于一个增广的上下文无关文法，还需要一个增广文法的新开始符号及其对应的开始符号产生式

2.2 Signature, Purpose Statement, Header and Functional Examples

State what kind of data the desired function consumes and produces. Formulate a concise answer to the question what the function computes. Define a stub that lives up to the signature.

根据这些类型的定义，我们足以定义出整个 LR(1) 语法分析程序的所有函数，如下：

closure

输入 LR(1) 项目集 I

输出 项目集的闭包 CLOSURE(I)

注 1 使用柯里化，这里不需要输入产生式表。

get-closure-function

输入 产生式的列表

输出 CLOSURE 函数

注 2 同理下面几个函数也使用柯里化，降低复杂度。

get-look-ahead

输入 产生式列表

输出 $look - ahead$ 函数

look-ahead

输入 LRItem

输出 当前 LRItem 在获取到下一个非终结符后的部分的 First 集

注 3 *若有* LRItem $= [A \to \alpha \cdot B\beta, a]$ *则* $look{-}ahead$(LRItem) $= FIRST(\beta)$*。这里不需要原来的 look-ahead 是因为在后续计算中加入了 look-ahead 的考量。*

go

输入 LR(1) 项目集闭包、syntax-item

输出 闭包在获取到该 syntax-item 后产生的新项目集 (非闭包)

look-ahead-and-reduce

输入 LR(1) 项目集闭包、(读头下的非终结符对应的)syntax-item

输出 闭包中可以执行规约的项目列表

注 4 *这是我实现的方法和书上方法主要的不同：在龙书中，LR 自动机的一个项目实则对应了一个项目集 (闭包)，所以我们无需对于状态进行编号，而只需要通过闭包的转换即可实现等价的自动机行为。这里输出的的可规约的项即为在预测分析表中填上的 r，而使用的产生式可以直接通过该规约项所对应的产生式计算得到。*

注 5 *实际上，我们不需要返回一个列表，因为 LR(1) 不产生规约-规约冲突。*

try-to-shift-in

输入 LR(1) 项目集闭包、(读头下的非终结符对应的)syntax-item

输出 闭包中可以执行移进的项目列表

try-to-shift-in

输入 LR(1) 项目集闭包、(读头下的非终结符对应的)syntax-item

输出 闭包中可以执行移进的项目列表

build-lr1-automata

输入 产生式列表、增广文法的开始符号的产生式、EOF 符号

输出 可以在一个 token 流上执行的自动机

注 6 *对应于龙书中的 LR(1) 算法主体，也就是所谓的 LR(1) 分析的主控。*

2.2.1 LR 自动机

从实现上说，实际上这不是一个自动机，而是一个先柯里化后的递归函数！

假设我们有一个词法单元（Token，记为 T）组成的列表 $t_1, t_2, \cdots, t_n, \#$，LR(1) 语法分析的输出应该是一颗语法树。简单说即为：

$$f_1 : T \times T \times \cdots \times T \to \{\text{Syntax Tree}\}$$

使用柯里化后：

$$\begin{aligned} f_1 :& t_1 \mapsto f_2 \\ f_2 :& t_2 \mapsto f_3 \\ & \cdots \\ f_n :& t_n \mapsto f_{n+1} \\ f_{n+1} :& \# \mapsto \text{Syntax Tree} \end{aligned}$$

可以看出，其输入都是相同的，而提供给函数的实际上是一个流对象（生成器），所以考虑最后将其转换为递归函数实现。在实验 1 中，已经说明了：状态机的实质就是从函数和输入到另一个函数的映射（状态转换函数）。因此，在语法分析实现过程中，将 LR 自动机实现为这样的函数。但实现中不难发现一个问题，由于我们在编码时无法确定状态总数，状态转换函数都定义为匿名函数。也就是说，这个递归函数跳转时跳转的对象是没有事先绑定的，显然，这是一个 Y-组合子问题。

当然，我们也可以做出一点改变，用一个很不函数式的方法来实现类似于一个 Y-组合子的方法，我们将每一个状态及其对应的状态转换函数，写入一个哈希表中，在进行状态跳转的同时在哈希表里面查询相应的表项执行，来避免 Y-组合子中无法进行匿名函数变量绑定的问题。

2.3 Function Template

Translate the data definitions into an outline of the function.

在2.3中，仅仅给出 `build-lr1-automata` 的定义过程，因为该函数足以诠释整个算法的构造流程。

2.3.1 LR(1) 自动机构造基础

给定产生式列表、增广文法的开始符号产生式，我们可以计算如下内容：

1. CLOSURE 函数 (Currying)
2. I_0：即自动机的初始状态所对应的项集闭包
3. GOTO 函数

注 7 *在这里的 GOTO 函数和先前定义的 GO 函数是不同的，GO 函数仅仅计算出从闭包 I_1 通过输入文法符号 A 得到的下一个闭包 I_2，但在这里，I_2 是通过 I_1 推导的，虽然本质上重复两次计算 GO(I_1, A) 所得到的 I_2 是相同的（函数是单值映射），但在计算机内部，由于通过了两次推导，用面向对象的说法即是两个截然不同的对象。*

注意，上面实现的 GOTO 函数是可以接受终结符号的！在报告实现的语法分析器中，由于将自动机的状态和 LR(1) 项目集构建了一一对应关系，即自动机运行时能确定自身状态对应的 LR(1) 项目集，可

以通过该项目集和输入的符号和 GOTO 函数来确定执行移进或者规约。换言之，不需要用 ACTION 函数即可实现该自动机。实现如下：考虑原有的移进和规约的执行逻辑，

在 LR(1) 分析表中填入移进，当下面两项中有且仅有一项成立：

1. $\mathrm{GO}(I_1, a) = I_2$ 且 a 是终结符号，那么在 ACTION 中填入移进并且转状态 I_2（项目集对应的状态）
2. $\mathrm{GO}(I_1, A) = I_2$ 且 A 是非终结符号，那么在 GOTO 中填入状态 I_2

重点关注非终结符号情况：在 LR 自动机执行的过程中，**必定是在规约得到了一个非终结符号后执行 GOTO 操作**。对于这类情况我们可以简单的执行如下三个内容：把读头下的内容转换为该非终结符号，并重新执行移进所对应的逻辑即可。

2.3.2 LR(1) 预测分析程序

这样的设计是合理的，我们可以设置这样的函数参数 `buf`，作为" 移入" 的符号。读头下的 `token` 它一定和某一个终结符匹配，那么转换为一个`matched-item`，其中的`stx`属性是其匹配的终结符，`content`对应该`token`。考虑两种情况：

1. 移入：程序在 token 流中取出下一个 token，并作为下一状态转换函数的 `buf` 参数传入。
2. 规约：`buf` 的符号需要保存（作为返回值传递给上一层递归函数调用），所以我们在规约前后都需要保存当前读头下的非终结符。

   在规约完后，该函数会返回`buf`和一颗语法树。（这相当于原来 LR 分析主控算法中的使用某个产生式来执行规约）然后我们对于拿到的语法树的头部非终结符执行移进（相当于按照分析表，执行 GOTO 部分），并将原`buf`中的词法单元执行传给该状态转换函数尝试进行下一步的状态转换。

   具体实现请见 5.1

2.3.3 优先级判断

不难发现，我们设计的二义文法在优先级上体现为移进-规约冲突，这可以通过：

- 规约的 LR1 项的产生式右部在 $\cdot$ 之前的部分的最后一个终结符
- 移进的 LR1 项的移进符号（实际上就是读头下的符号）

的优先级来判断！

考虑移进规约冲突如下：

$$(E \to E + E\cdot, \times) \quad (E \to E \cdot \times E, +)$$

由于规约项中的 $\times$ 比 $+$ 优先级高，所以执行移进。其余情况同理处理。

2.4 Function Definition

Fill in the gaps in the function template. Exploit the purpose statement and the examples.

这里就不多进行函数定义的解释了，具体实现方法可以在5找到。

3 算法测试

Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

我们用龙书的例 4.54 作为测试用例：

```
#lang racket

(require "types.rkt")
(require "syntax-define.rkt")
(require "build.rkt")
(require "../lex/lexer.rkt")
(require racket/generator)


(define STX-FAKE-S
  (syntax-item "<augmented>" 0 (void)))
(define STX-S
  (syntax-item "<S>" 0 (void)))

(define STX-C
  (syntax-item "<C>" 0 (void)))

(define STX-c
  (syntax-item "<c>" 0 (lambda (ch) (char=? #\c ch))))

(define STX-d
  (syntax-item "<d>" 0 (lambda (ch) (char=? #\d ch))))

(define STX-EOF (syntax-item "EOF" 1024 (lambda (tok) (void? tok))))

(define productions
  (list
   (production STX-FAKE-S (list STX-S))
   (production STX-S (list STX-C STX-C))
   (production STX-C (list STX-c STX-C))
   (production STX-C (list STX-d))))

(define aug
  (LRItem STX-FAKE-S (list) (list STX-S) STX-EOF))

(define gen
  (let ([s (string->list "cdccd")])
    (generator ()
               (let rec ([l s])
                 (if (empty? l)
                     (void)
                     (begin
                       (yield (first l))
                       (rec (rest l))))))))

(time
 (let ([automata (build-lr1-automata productions aug STX-EOF)])
   (let ([retval (automata gen)])
     (display-tree-mma retval))))

(printf "\n\n#################### RESULT OF LR[1] ####################\n")
```

该测试用例产生了输出：

```
(#(struct:syntax-item <S> 0 #<void>) #(struct:syntax-item <C> 0 #<void>) #(struct:
    syntax-item <augmented> 0 #<void>))
First[ <S> ] = {<c>, <d>, }
First[ <C> ] = {<c>, <d>, }
First[ <augmented> ] = {<c>, <d>, }
Found:  #(struct:syntax-item <d> 0 #<procedure:...cp/syntax/test3.rkt:22:23>)
Found:  #(struct:syntax-item <c> 0 #<procedure:...cp/syntax/test3.rkt:19:23>)
ITERATING NEXT...
        Item(<augmented> -> ^ <S> , EOF)
        Item(<S> -> ^ <C> <C> , EOF)
        Item(<C> -> ^ <c> <C> , <d>)
        Item(<C> -> ^ <c> <C> , <c>)
        Item(<C> -> ^ <d> , <d>)
        Item(<C> -> ^ <d> , <c>)
ITERATING NEXT...
        Item(<augmented> -> <S> ^ , EOF)
ITERATING NEXT...
        Item(<C> -> <d> ^ , <d>)
        Item(<C> -> <d> ^ , <c>)
ITERATING NEXT...
        Item(<C> -> <c> ^ <C> , <d>)
        Item(<C> -> <c> ^ <C> , <c>)
        Item(<C> -> ^ <c> <C> , <d>)
        Item(<C> -> ^ <d> , <d>)
        Item(<C> -> ^ <c> <C> , <c>)
        Item(<C> -> ^ <d> , <c>)
ITERATING NEXT...
        Item(<S> -> <C> ^ <C> , EOF)
        Item(<C> -> ^ <c> <C> , EOF)
        Item(<C> -> ^ <d> , EOF)
ITERATING NEXT...
        Item(<C> -> <c> <C> ^ , <d>)
        Item(<C> -> <c> <C> ^ , <c>)
ITERATING NEXT...
        Item(<C> -> <d> ^ , EOF)
ITERATING NEXT...
        Item(<C> -> <c> ^ <C> , EOF)
        Item(<C> -> ^ <c> <C> , EOF)
        Item(<C> -> ^ <d> , EOF)
ITERATING NEXT...
        Item(<S> -> <C> <C> ^ , EOF)
ITERATING NEXT...
        Item(<C> -> <c> <C> ^ , EOF)

Exiting...

Tree["<augmented>",{Tree["<S>",{Tree["<C>",{"[<c>]c", Tree["<C>",{"[<d>]d"}]}], Tree["<C>"
    ,{"[<c>]c", Tree["<C>",{"[<c>]c", Tree["<C>",{"[<d>]d"}]}]}]}]}]cpu time: 0 real time
    : 4 gc time: 0



################## RESULT OF LR[1] ##################
```

注 8 前几行是从产生式中推断所有的非终结符的 *First* 集，然后得到了所有的终结符号，继而通过不动点算法迭代产生所有的闭包。最后通过自动机来实现语法分析。

通过最后一行的输出，放在 Mathematica 中进行图形化：

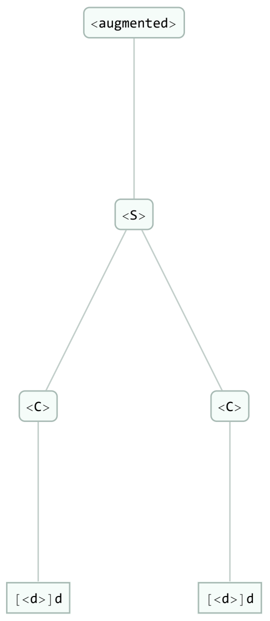

图 1: 对于 dd 的分析结果

我们将原来的测试字符串替换为：`cdccd`，那么

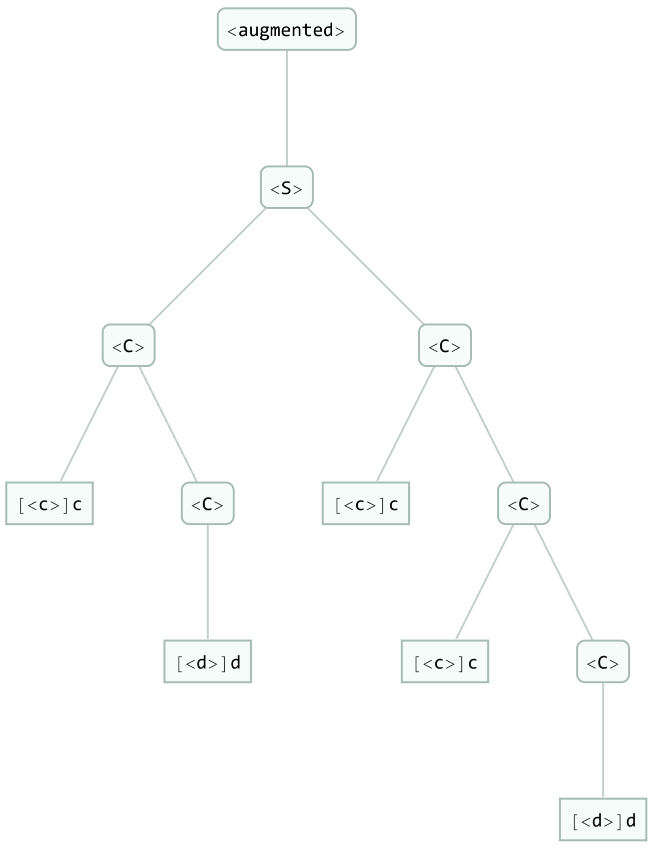

图 2: cdccd 的分析结果

当然，我们写出如下的产生式：

```scheme
(define productions
 (list
  (production STX-FAKE-S (list STX-EXPR))
  (production STX-EXPR (list stx-sep-lbracket STX-EXPR stx-sep-rbracket))
  (production STX-EXPR (list STX-EXPR stx-relop-plus STX-EXPR))
  (production STX-EXPR (list STX-EXPR stx-relop-minus STX-EXPR))
  (production STX-EXPR (list STX-EXPR stx-relop-divide STX-EXPR))
  (production STX-EXPR (list STX-EXPR stx-relop-assign STX-EXPR))
  (production STX-EXPR (list STX-EXPR stx-relop-multi STX-EXPR))
  (production STX-EXPR (list stx-number))
  (production STX-EXPR (list stx-identifier))))
```

$$1 - 2 + 3 * 4 - 5/6 + (7 + 8) * 9$$

也可以简单的对于下述表达式进行语法分析:

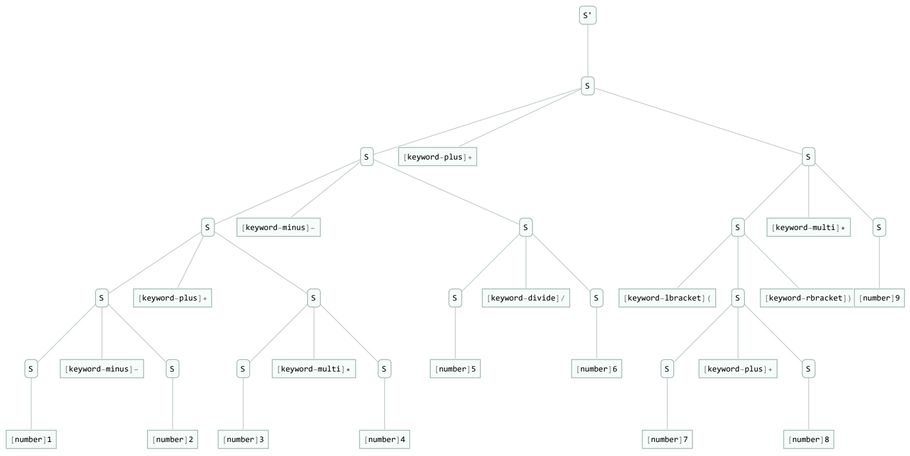

图 3: 上述表达式的语法树。可以看出，正确的进行了 * 、+ 、() 的优先级判断.

当然也可以对下述程序进行分析:（产生式在：5.2）

```pascal
function func(a : int) : int
var
    x : int[10];
begin
    id123 := 2 * (3 + 4 * 5) * 5 - 5 / 6;
    2 * 3 + 4 - 5;
end

program prog;
var
    x : int;
begin
    id123 := 2 * (3 + 4 * 5) * 5 - 5 / 6;
    2 * 3 + 4 - 5;
    fun(1, 2);
end
```

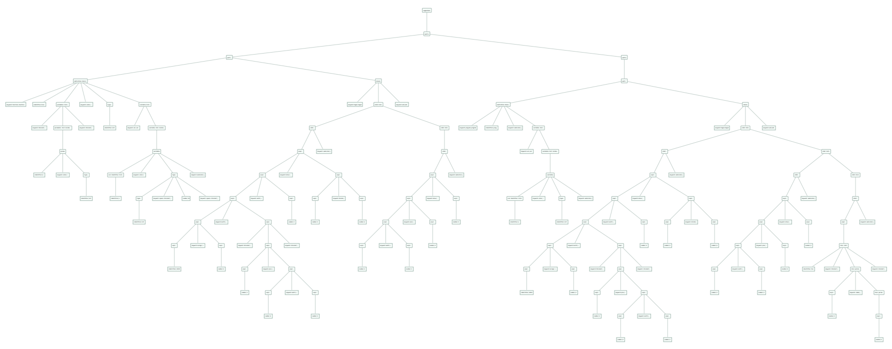

图 4: 上述程序的语法分析结果

注 9 这个放大应该能看清楚吧。(毕竟是 *pdf* 导出的嘿嘿) *:)*

4 总结

妈的，终于调通了！总的来说，实验对于书上各个算法进行一些等价变换，然后给出了函数签名，阐述函数作用，最终完成这样的一个算法实现。当然中间 Debug 真的很复杂，因为要将状态转换表和 LR(1) 项集在脑子里面不断地转换，将 reduce 操作的退栈过程等价到函数调用返回时的返回、退栈操作，把 GOTO 和 ACTION 整合，并实现原有的功能，最终实现了 LR(1) 的自动推导、和自动机分析，能够配合原有的词法分析器实现对于简单 Pascal 源码的词法、语法分析，产生对应的语法树。

5 源码

两次实验的报告和源码都可以在我的gitee repo中找到。

5.1 build-lr1-automata

Listing 2: build.rkt

```
#lang racket

(require "types.rkt")
(require "closure.rkt")
(require "goto.rkt")
(require data/queue)

(define (look-ahead-and-reduce lritem-list sym)
  (for/list ([lri lritem-list]
             #:when(and (empty? (LRItem-right-tail lri))
                        (eq? sym (LRItem-look-ahead lri))))
    lri))

(define (try-to-shift-in lritem-list sym)
  (for/list ([lri lritem-list]
             #:when(and (not (empty? (LRItem-right-tail lri)))
                        (eq? sym (first (LRItem-right-tail lri)))))
    lri))

(define (get-all-terminals prod)
  (set->list(list->set(filter terminal? (flatten (for/list ([p prod]) (production-right p)
      ))))))

(define (stx-can-go item-list)
  (set->list
   (list->set
    (for/list
        ([i item-list]
         #:when (not (empty? (LRItem-right-tail i))))
      (first (LRItem-right-tail i))))))

(define (costom-list-member? eqf lst item)
  (if (empty? lst)
      #f
      (if (eqf item (first lst))
          #t
          (costom-list-member? eqf (rest lst) item))))

(define (build-lr1-automata prod-list augmented-item eof-syntax)
  (define closure (get-closure-function prod-list))

  (define I0 (closure (list augmented-item)))

  (define term->stx-item
    (let ([all-terminals (get-all-terminals prod-list)])
      (for-each (lambda (t) (printf "Found:\t~a\n" t)) all-terminals)
      (lambda (tok)
        (if ((syntax-item-matcher eof-syntax) tok)
            eof-syntax
            (let ([stxl (for/list
                          ([t all-terminals]
                           #:when((syntax-item-matcher t) tok))
                          t)])
              (if (empty? stxl)
                  (printf "ERROR: cannot identify this terminal.~a\n" tok)
                  (first stxl)))))))

  (define (generate-closure-from item-list)
    (for/list ([stx (stx-can-go item-list)])
      (closure (go item-list stx))))

  (define (find-not-in clist c)
    (for/list ([clos (generate-closure-from c)]
               #:when(not(costom-list-member? closure-eq? clist clos)))
      clos))

  (define all-closures (let ([q (make-queue)])
                         (enqueue! q I0)
                         (let expand-closure ([result (list)])
                           (if (queue-empty? q)
                               result
                               (let* ([top (dequeue! q)])
                                 (printf "ITERATING NEXT...\n")
                                 (for-each (lambda (i)
                                             (printf "\t")
                                             (display-lritem i)) top)
                                 (set! result (append result (list top)))
                                 (for-each
                                  (lambda (g) (enqueue! q g))
```

```
                                (find-not-in result top))
                               (expand-closure result))))))

(define goto
  (let ([ht (make-hash)])
    (for-each (lambda (c) (hash-set! ht c (make-hash))) all-closures)
    (for*/list ([c1 all-closures]
                [c2 all-closures])
      (for/list ([stx (stx-can-go c1)]
                 #:when(closure-eq? (closure (go c1 stx)) c2))
        (hash-set! (hash-ref ht c1) stx c2)))
    (lambda (cl stx) (hash-ref (hash-ref ht cl) stx))))

(define (work-on-generator gen)
  (let ([ht (make-hash)]
        [sym (gen)])
    (for-each
     (begin
       (lambda (c)
         (hash-set!
          ht c
          (lambda (sym)
            (let redo ([buffer sym])
              (printf "Entering state(closure):\n")
              (for-each (lambda (c) (display "\t- ") (display-lritem c)) c)
              (define buf (if (or (syntax-item? buffer)
                                  (pair? buffer)) buffer (term->stx-item buffer)))
              (printf "Received a symbol [[")
              (if (pair? buf)
                  (printf "~a" (syntax-item-id (syntax-tree-node-head (car buf))))
                  (printf "~a" (syntax-item-id buf)))
              (displayln "]]\n\tCurrent Closure is \n")
              (for-each (lambda (c) (display "\t- ") (display-lritem c)) c)
              (define (shift-in-helper retval matched-stx transformer)
                (let ([prod-in-use (cdr (car (car retval)))]
                      [prod-head (car (car (car retval)))]
                      [stack (cdr (car retval))]
                      [next-symbol-in-buffer (cdr retval)])

                  (printf "Reducing... \n")
                  (for-each (lambda (c) (display "\t- ") (display-lritem c)) c)

                  (define stack-added (append
                                       stack
                                       (list (transformer
                                              matched-stx
                                              prod-in-use))))

                  (printf "\tcurrently in stack is\n\t[")
                  (for-each
                   (lambda (item)
                     (printf "\n\t\t- ")
                     (display-tree item))
                   (reverse stack-added))
                  (printf "\n\t]\n")

                  (if (eq? (length prod-in-use) 1)
                      ; reduce complete!
                      ; 当前的CLOSURE中一定有一个能够处理当前的syntax-node 对应的非终结
            符 （即执行了goto操作）
        (let ([final-tree (syntax-tree-node prod-head (reverse stack-added))])
          (printf "Reduce done!!! now goto another state...(redo) Tree is\n\t")
          (display-tree final-tree)
          (printf "\n")
          (printf "From state:\n")
          (for-each (lambda (c) (display "\t- ") (display-lritem c)) c)
          (printf "To..")
          (redo (cons final-tree next-symbol-in-buffer)))
        ; reduce 还没有结束，继续退栈
        (begin
          (printf "\tReduce not complete : prod left is ~a\n" prod-in-use)
          (cons (cons (cons prod-head (rest prod-in-use))
                      stack-added)
                next-symbol-in-buffer)))))
(if (pair? buf)
    (let ([nt-tree-node (car buf)]
          [next-symbol-in-buf (cdr buf)])
      (printf "Here reduction complete when look-ahead is ~a\nAlso for the tree...\n\t" next-symbol-in-buf)
      (begin (display-tree nt-tree-node) (displayln ""))
      (printf "Current state is\n")
      (for-each (lambda (c) (display "\t- ") (display-lritem c)) c)
      (if (eq? (syntax-tree-node-head nt-tree-node) (LRItem-left augmented-item))
          (begin
            (printf "All done!!!\n") ; Augmented item
            (display-tree nt-tree-node)
            (printf "\n\nExiting...\n\n")
            nt-tree-node)
          (let* ([next-state (hash-ref ht (goto c (syntax-tree-node-head nt-tree-node)))]
                 [retval (next-state next-symbol-in-buf)])
            (shift-in-helper retval nt-tree-node (lambda (m p) m)))))
    (let ([item-can-reduce (look-ahead-and-reduce c buf)]
          [item-can-shiftin (try-to-shift-in c buf)])
      ; 接收到一个终结符号，执行移进或者规约
      (printf "\tItem->Reduce:~a\n" (length item-can-reduce))
      (printf "\tItem->Shift: ~a\n" (length item-can-shiftin))
      (define (reduce)
        (if (< 1 (length item-can-reduce)); reduce to a list and do nothing.
            (displayln "Error: cannot decide which prod to reduce")
            (let ([item (first item-can-reduce)])
              ; return the production
              (printf "Reduce! (look-ahead = ~a) use item: \n\t" buf)
              (display-lritem item)
              (cons
               (cons
                (cons
                 (LRItem-left item)
                 (reverse (LRItem-right-head item)))
                (list))
               buffer))))
      (define (shift-in)
        (let* ([matched-stx buf]
               [next-state (hash-ref ht (goto c matched-stx))])
```

```
                          (printf "Shift in! \n\tMatched<~a>-stx is ~a \n" buffer matched-stx)
                          (let([retval (next-state (gen))])
                            (if (syntax-tree-node? retval)
                                (display-tree retval)
                                (void))
                            (shift-in-helper
                             retval
                             matched-stx
                             (lambda (matched-stx prod-in-use)
                               (syntax-tree-node
                                (first prod-in-use)
                                (matched-item matched-stx buffer)))))))
                      (if (empty? item-can-shiftin)
                          (if (empty? item-can-reduce)
                              (begin
                                (displayln "Error. nothing to do...")
                                (void)) ;error.
                              (reduce))
                          (if (empty? item-can-reduce)
                              ; shift in the terminal and go to another state.
                              ; 执行 移进 + 转下一状态
                              (shift-in)
                              (begin ; not clear -> decide on priority! or error here.
                                (displayln "Cannot decide which to use (RS).\n reduce is\n")
                                (for-each display-lritem item-can-reduce)
                                (printf "\nshift-in is\n")
                                (for-each display-lritem item-can-shiftin)
                                (if (eq? 1 (length item-can-reduce))
                                    (let ([head (LRItem-right-head (first item-can-reduce))])
                                      (if (>= (length head) 2)
                                          (if (syntax-item? (second (reverse head)))
                                              (begin
                                                (printf "But I found that I can decide by syntax ~a\n"
                                                        (second(reverse head)))
                                                (let ([prio-shift-in (syntax-item-priority (first (LRItem-right-tail (first item-can-shiftin))))]
                                                      [prio-reduce (syntax-item-priority (second(reverse head)))])
                                                  (if (> prio-shift-in prio-reduce)
                                                      (shift-in)
                                                      (reduce))))
                                              (void)) ; error
                                          (void)))    ; error
                                    (void)))))))))))) ; error
       all-closures)
      ((hash-ref ht I0) sym)))
  work-on-generator)

(provide build-lr1-automata)
```

5.2 Pascal 产生式定义

Listing 3: main.rkt

```
#lang racket

(require "types.rkt")
(require "syntax-define.rkt")
(require "build.rkt")
(require "../lex/lexer.rkt")

; Lexical Analysis
(define file-to-analysis
  (command-line
   #:program "Lexical Analyser"
   #:args ([filename "test1.pas"])
   filename))

(define content
  (bytes->string/utf-8 (file->bytes file-to-analysis)))

(displayln content)

(define gen (lexical-generator content))


; Non-terminate symbols
(define STX-FAKE-S
  (syntax-item "<augmented>" 0 (void)))

(define STX-EXPR
  (syntax-item "<expr>" 0 (void)))

(define STX-EOF (syntax-item "EOF" 1024 (lambda (tok) (void? tok))))

(define STX-PARTS
  (syntax-item "<parts>" 0 (void)))

(define STX-PART
  (syntax-item "<part>" 0 (void)))

(define STX-PROGRAM
  (syntax-item "<program>" 0 (void)))

(define STX-BLOCK
  (syntax-item "<block>" 0 (void)))

(define STX-STMT-LIST
  (syntax-item "<stmt-list>" 0 (void)))

(define STX-STMT
  (syntax-item "<stmt>" 0 (void)))

(define STX-BRANCH
  (syntax-item "<branch>" 0 (void)))

(define STX-WHILE-LOOP
  (syntax-item "<while-loop>" 0 (void)))

(define STX-DEFINITION
  (syntax-item "<definition-block>" 0 (void)))

```

```
(define STX-PARAM-LIST
  (syntax-item "<parameter-list>" 0 (void)))

(define STX-PARAM-LIST-INSIDE
  (syntax-item "<parameter-list-inside>" 0 (void)))
(define STX-PARAM
  (syntax-item "<param>" 0 (void)))

(define STX-VAR-LIST
  (syntax-item "<variable-list>" 0 (void)))

(define STX-VAR-LIST-INSIDE
  (syntax-item "<variable-list-inside>" 0 (void)))

(define STX-VAR-ITEM
  (syntax-item "<variable>" 0 (void)))

(define STX-VAR-NAME-LIST
  (syntax-item "<var-identifier-list>" 0 (void)))

(define STX-FCALL
  (syntax-item "<func-call>" 0 (void)))

(define STX-TYPE
  (syntax-item "<type>" 0 (void)))

(define STX-FCALL-PARAM
  (syntax-item "<func-param>" 0 (void)))


(define productions
  (list
   (production STX-FAKE-S (list STX-PARTS))
   (production STX-PARTS (list STX-PART))
   (production STX-PARTS (list STX-PART STX-PARTS))
   (production STX-PART (list STX-DEFINITION STX-BLOCK))
   (production STX-TYPE (list stx-identifier))
   (production STX-TYPE (list
                         STX-TYPE
                         stx-sep-square-lbracket
                         stx-number
                         stx-sep-square-rbracket))

   ; definition block can be function / program / procedure

   ; Functions:
   (production STX-DEFINITION (list stx-keyword-function
                                    stx-identifier
                                    STX-PARAM-LIST
                                    stx-sep-colon
                                    STX-TYPE
                                    STX-VAR-LIST))
   (production STX-DEFINITION (list stx-keyword-function
                                    stx-identifier
                                    STX-PARAM-LIST
                                    stx-sep-colon
                                    stx-identifier))

   ; main program
   (production STX-DEFINITION (list stx-keyword-program
                                    stx-identifier
                                    stx-sep-semicolon
                                    STX-VAR-LIST))

   ; Parameter list:
   (production STX-PARAM-LIST (list stx-sep-lbracket stx-sep-rbracket))
   (production STX-PARAM-LIST (list stx-sep-lbracket STX-PARAM-LIST-INSIDE
       stx-sep-rbracket))
   (production STX-PARAM-LIST-INSIDE (list STX-PARAM stx-sep-comma STX-PARAM-LIST-INSIDE))
   (production STX-PARAM-LIST-INSIDE (list STX-PARAM))
   (production STX-PARAM (list stx-identifier stx-sep-colon STX-TYPE))

   ; Variable list:
   (production STX-VAR-LIST (list stx-keyword-var STX-VAR-LIST-INSIDE))
   (production STX-VAR-LIST-INSIDE (list STX-VAR-ITEM STX-VAR-LIST-INSIDE))
   (production STX-VAR-LIST-INSIDE (list STX-VAR-ITEM))
   (production STX-VAR-ITEM (list STX-VAR-NAME-LIST stx-sep-colon STX-TYPE
       stx-sep-semicolon))
   (production STX-VAR-NAME-LIST (list stx-identifier))
   (production STX-VAR-NAME-LIST (list stx-identifier STX-VAR-NAME-LIST))

   ; Definition for a block of program. (or just one statement)
   (production STX-BLOCK (list stx-keyword-begin STX-STMT-LIST stx-keyword-end))
   (production STX-BLOCK (list stx-keyword-begin stx-keyword-end))
   (production STX-BLOCK (list STX-STMT))
   ; Definition for statements.
   (production STX-STMT (list STX-EXPR stx-sep-semicolon))
   (production STX-STMT-LIST (list STX-STMT))
   (production STX-STMT-LIST (list STX-STMT STX-STMT-LIST))

   ; Definition for if ... then ... else ...
   (production STX-STMT (list STX-BRANCH))
   (production STX-BRANCH(list stx-keyword-if STX-BLOCK))
   (production STX-BRANCH(list stx-keyword-if STX-BLOCK stx-keyword-else STX-BLOCK))

   ; Definition for while loop
   (production STX-STMT (list STX-WHILE-LOOP))
   (production STX-WHILE-LOOP (list stx-keyword-while stx-sep-lbracket STX-EXPR
                                    stx-sep-rbracket STX-BLOCK))

   ; Definition for expressions
   (production STX-EXPR (list stx-sep-lbracket STX-EXPR stx-sep-rbracket))
   (production STX-EXPR (list STX-EXPR stx-relop-plus STX-EXPR))
   (production STX-EXPR (list STX-EXPR stx-relop-minus STX-EXPR))
   (production STX-EXPR (list STX-FCALL))
   (production STX-EXPR (list STX-EXPR stx-relop-divide STX-EXPR))
   (production STX-EXPR (list STX-EXPR stx-relop-assign STX-EXPR))
   (production STX-EXPR (list STX-EXPR stx-relop-multi STX-EXPR))
   (production STX-EXPR (list stx-number))
   (production STX-EXPR (list stx-identifier))
   (production STX-FCALL (list stx-identifier stx-sep-lbracket STX-FCALL-PARAM
       stx-sep-rbracket))
   (production STX-FCALL (list stx-identifier stx-sep-lbracket stx-sep-rbracket))
   (production STX-FCALL-PARAM (list STX-EXPR stx-sep-comma STX-FCALL-PARAM))
   (production STX-FCALL-PARAM (list STX-EXPR))
   ))

```

```
(define aug
  (LRItem STX-FAKE-S (list) (list STX-PARTS) STX-EOF))

(time
 (let ([automata (build-lr1-automata productions aug STX-EOF)])
   (let ([retval (automata gen)])
     (printf "Content of ~a is:\n~a\n#################### RESULT OF LR[1]
         ####################\n\n" file-to-analysis content)
     (display-tree-mma retval))))

(printf "\n\n#################### RESULT OF LR[1] ####################\n")
```