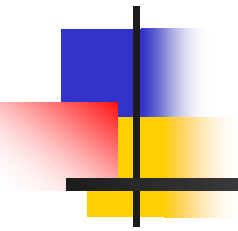
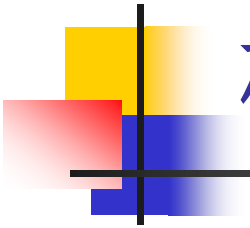


句法抽象：syntax-case展开器

《代码之美》第25章

信息学低年级讨论班 青慈阳
00801111



概述

- 本章中作者主要讨论了在宏展开中出现的变量捕获问题，提出了**Scheme**中一种保证健康宏展开的实现——**syntax-case**系统，并给出了一个简化版本的展开器

问题背景(I)

- 在我们编程时，某些模式会反复出现，将其提取出来可以使程序更简洁易读

- 但是其中有些模式只在某类或某个特定程序中出现，语言设计者未必能预见，即使预见到了，设计者也未必会把这些模式放在语言的核心部分

问题背景(II)

- 对此，设计者通常的做法是在语言里加入句法抽象的机制。例如C语言的预处理宏，Common Lisp的宏，及一些外部工具等等

- 本章处理的是Lisp中的宏，它基于表达式的替换，替换表达式在Lisp系统内运算出来

问题的产生——变量捕获

- 由于宏机制的实质是一些简单的替换，因此尤其要注意被替换部分与外部的独立性、隔绝性，否则就会出现一些难以预料的问题

- 一类情况是宏调用子式中的标识符在展开时因作用域不当而引发的变量捕获问题

两个例子(I)

- 下面把**Scheme**的**or**表达式变换为**let**和**if**的简单代码：

  (or e1 e2) -> (let ([t e1]) (if t t e2))

- 如果**or**表达式第一项子式为真，就返回第一项子式的值，否则返回第二项子式的值，用**let**来给**e1**命名，以免计算两次

两个例子(II)

- 上述变换大多数情况下都会正常工作，但是如果标识符**t**在**e2**内是自由变量（即**t**在**e2**内没有绑定），那么运算结果就会出错，例如表达式

  (let ([t #t]) (or #f t)) 将被展开为

  (let ([t #t])
  (let ([t #f])
  (if t t t))) 从而得到错误的值

两个例子(III)

- 上述的变量捕获是源于宏内部的变量绑定，这总能通过重新生成绑定用的标识符来解决（如上例中把**t**换成**g**即可）

- 但是对于因宏内部的变量引用造成的捕获，就未必能用上述方案解决了

两个例子(IV)

- 例如：

  (let ([*if* (lambda (x y z) “oops”)])

  (or #f #f)) 将被展成

  (let ([*if* (lambda (x y z) “oops”)])

  (let ([t #f]) (*if* t t #f)))

  注意此处展开后的**if**已经与**or**的宏定义中的**if**意义不一样了，输出自然也不同

两个例子(V)

- 显然，对上面这个例子，如果把**let**、**if**等作为保留字，是能解决问题，但这明显不是通用的办法（问题产生的根源并未被消除）

- 解决上述问题的办法是在宏的外部修改标识符的名字而不是在内部

两个例子（小结）

- 综上所述，宏展开的变量捕获问题的来源主要有两类——变量绑定和变量引用

- 问题的本质在于宏的内外部没有做到隔离，因此健康的宏展开必须实现源代码上的词法作用域，为此可以不惜对源代码的标识符重命名以达到隔离的效果

Syntax-case系统

- 上面提到的问题以及解决它的思想、方法，是若干学者研究的成果
- **Kohlbecker**等人提出了健康宏展开算法（**KFFD**）
- **Chinger**等人开发了**syntax-rules**系统
- 本章介绍的**syntax-case**系统既是一种改进，又只是一个简化版（篇幅所限）

Syntax-case实例简介(I)

- 仍以前面**or**的宏定义为例，在**syntax-case**中，其定义如下：

```
(define-syntax or
   (lambda (x)
     (syntax-case x ()
        [ (_ e1 e2)
          (syntax (let ([t e1]) (if t t e2)))])))
```

Syntax-case实例简介(II)

- 形式**define-syntax**创建关键词绑定，这里把关键词**or**绑定到一个变换过程（变换器）上。在展开时通过计算右边的**lambda**表达式来得到变换器

- 形式**syntax-case**用于解析输入，而**syntax**则用于构造输出（利用模式匹配）

Syntax-case实例简介(III)

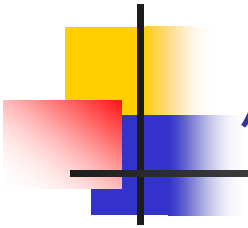
- 形式(syntax template)可以简写为#’template

- 宏可以通过letrec-syntax绑定到单个表达式里（与scheme中define和let的用法关系相对应，也即宏的定义可以是局部的）

Syntax-case实例简介(IV)

- 宏可以递归，例如**or**的处理任意多子式的宏定义：

```
(define-syntax or
 (lambda (x)
  (syntax-case x () [(_) #’ #f] [(_ e) #’ e]
     [(_ e1 e2 e3 …)
      #’(let ([t e1]) (if t t (or e2 e3 …)))])))
```



展开算法(I)

- **Syntax-case**的展开算法是在输入表达式的抽象表示上操作，而不是在传统的符号表达式的表示上操作

- 抽象表示中封装了输入形式和用于决定标识符作用域的包装（**wrap**），包装则由记号和替代器组成

展开算法(II)

- 替代器在编译环境的帮助下把标识符映射到绑定。每当遇到绑定形式（例如**lambda**）时，替代器就被创建并加入句法对象的包装里。这些句法对象代表了绑定形式的绑定作用内的形式

展开算法(III)

- 展开以递归且自顶向下的方式进行。展开器遇到宏调用，就对遇到的形式调用相关的变换器，把它标上新建记号，然后再用同样的记号标注一次。同样的记号抵消了，所以只有宏生成的输出部分（也即不是从输入拷贝到输出的部分）会保留记号

展开算法(IV)

- 当展开器遇到核心形式时，就会生成展开器的输入语言里的一项核心形式。在生成过程中，每项子式在需要时递归展开，变量应用通过替换机制被生成的名字取代

源代码的表示(I)

- 如上所述，在**syntax-case**系统中，源代码被表示为抽象的句法对象：

```
(define-record syntax-object (expr wrap))
```

- 这里**define-record**形式创造了一种带特定名字的值和字段，以及一套操作它们的过程

源代码的表示(II)

- 这些过程包括：

make-syntax-object

syntax-object?

syntax-object-expr

syntax-object-wrap

它们的作用不难理解

源代码的表示(III)

- 每个包装由一组记号和替换器组成
- 记号由它们的对象身份来区别，无需任何字段　(define-record mark ())
- 替换器把符号名和一组记号映射到一条标签
- (define-record subst (sym mark* label))

源代码的表示(IV)

- 标签和记号一样用身份区分，无需字段
  (define-record label ())

- 由展开器维护的展开时环境将标签映射到绑定上。环境的结构就是传统的关联表。表中每个点对的**cons**是一个标签，**cdr**包含一个绑定。

源代码的表示(V)

- 绑定包含一个类型和一个值

  (define-record binding (type value))

- 类型决定绑定的性质：**macro**对应关键词绑定，**lexical**对应语词变量的绑定；值对应的是所需要的相应信息，例如关键词绑定所需的变换过程

生成展开器输出(I)

- 展开器输出的是属于核心语言的简单符号表达式，大部分时候用**quasiquote**句法创建。例如一个**lambda**表达式可以用形式参数**var**和函数体**body**创建如下：

  `(lambda (,var) ,body)

生成展开器输出(II)

- 展开器通过辅助函数**gen-var**创建新名字，同时维护一个本地的序列计数器

```
(define gen-var
 (let ([n 0])
  (lambda (id)
   (set! n (+ n 1))
   (let ([name (syntax-object-expr id)])
   (string->symbol (format “~s.~s” name n))))))
```

剥离句法对象(I)

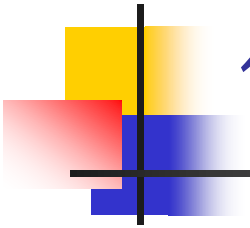
- 展开器用**strip**过程剥离所有内嵌句法对象和包装

```
(define (strip x)
 (cond [(syntax-object? x)
          (if (top-marked? (syntax-object-wrap x))
          (syntax-object-expr x)
          (strip (syntax-object-expr x)))]
         [(pair? x)
          (let ([a (strip (car x))] [d (strip (cdr x))])
          (if (and (eq? a (car x)) (eq? d (cdr x)))
              x  (cons a d)))]
         [else x]))
```

剥离句法对象(II)

- 其中

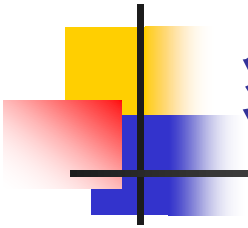
```
(define top-mark (make-mark) )

(define (top-marked? wrap)
   (and (not (null? wrap) )
        (or (eq? (car wrap) top-mark)
            (top-marked? (cdr wrap)))))
```



句法错误

- 展开器通过**syntax-error**报句法错误

```
(define (syntax-error object message)
  (error #f "~a ~s" message
         (strip object) ))
```



结构谓词

- 谓词**identifier?**确定一个句法对象是否代表标识符：

```
(define (identifier? x)
  (and (syntax-object? x)
       (symbol? (syntax-object-expr x))))
```

- 剥离句法对象后，谓词**self-evaluating?**判断该句法对象是否代表常数

```
(define (self-evaluating? x)
   (or (boolean? x) (number? x) (string? x) (char? x) ))
```

创建包装(I)

- 我们通过扩展包装在句法对象里加入记号或替换器

```
(define (add-mark mark x)
    (extend-wrap (list mark) x ))
(define (add-subst id label x)
    (extend-wrap
      (list (make-subst
              (syntax-object-expr id )
              (wrap-marks (syntax-object-wrap id)
              label))
        x))
```

创建包装(II)

- 扩展包装的定义如下：

```
(define (extend-wrap wrap x)
 (if (syntax-object? x)
   (make-syntax-object
     (syntax-object-expr x)
     (join-wraps wrap (syntax-object-wrap x)))
   (make-syntax-object x wrap) ) )
```

创建包装(III)

- 合并包装只要注意把相同的**mark**消掉即可

```
(define (join-wraps wrap1 wrap2)
  (cond [(null? wrap1) wrap2]  [(null? wrap2) wrap1]
   [else (let f ([w (car wrap1)] [w* (cdr wrap1)])
           (if (null? w*)
               (if (and (mark? w) (eq? (car wrap2) w) )
                   (cdr wrap2)
                   (cons w wrap2))
               (cons w (f (car w*) (cdr w*)))))]))
```

操纵环境

- 环境把标签映射到绑定上，并且用关联表表示。因此扩展环境需要加入一个单对：

```
(define (extend-env label binding env)
  (cons (cons (label binding) env)))
```

标识符解析(I)

- 确定与标识符联系的绑定是一个两步过程。第一步确定标识符包装里与该标识符联系的标签，第二步是在当前环境里查询该标签：

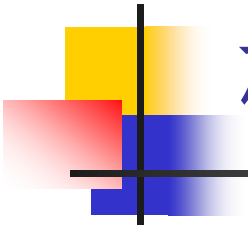
```
(define (id-binding id r)
    (label-binding id (id-label id) r))
```

标识符解析(II)

```
(define (id-label id)
  (let ([sym (syntax-object-expr id)]
        [wrap (syntax-object-wrap id)])
    (let search ([wrap wrap] [mark* (wrap-marks wrap)])
      (if (null? wrap)   (syntax-error id "undefined identifier")
          (let ([w0 (car wrap)])
            (if (mark? w0)
                (search (cdr wrap) (cdr mark*))
                (if (and (eq? (subst-sym w0) sym)
                         (same-marks? (subst-mark* w0) mark*))
                         (subst-label w0)
                         (search (cdr wrap) mark*))))))))
```

标识符解析(III)

```
(define (wrap-marks wrap)
    (if (null? wrap) '()
        (let ([w0 (car wrap)])
            (if (mark? w0)
                (cons w0 (wrap-marks (cdr wrap)))
                (wrap-marks (cdr wrap))))))
(define (same-marks? m1* m2*)
    (if  (null? m1*)   (null? m2*)
         (and (not (null? m2*))
                (eq? (car m1*) (car m2*) )
                (same-marks? (cdr m1*) (cdr m2*)))))
```

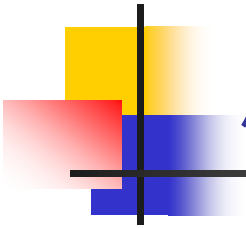


标识符解析(IV)

```
(define (label-binding id label r)
 (let ([a (assq label r)])
   (if a  (cdr a)
      (syntax-error id "displaced lexical"))))
```

展开器(I)

```
(define (exp x r mr)
 (syntax-case x ( )
  [id  (identifier? #'id)
      (let ([b (id-binding #'id r)])
           (case (binding-type b)
               [(macro) (exp (exp-macro (binding-value b) x) r mr)]
               [(lexical) (binding-value b)]
               [else (syntax-error x "invalid syntax")]))]
  [(e0 e1 ...)   (identifier? #'e0)
      (let ([b (id-binding #'e0 r)])
           (case (binding-type b)
              [(macro) (exp (exp-macro (binding-value b) x) r mr)]
              [(lexical) `(,(binding-value b) ,@(exp-exprs #'(e1 ...) r mr))]
              [(core) (exp-core (binding-value b) x r mr)]
              [else (syntax-error x "invalid syntax")]))]
  [(e0 e1 ...)  `(,(exp #'e0 r mr) ,@(exp-exprs #'(e1 ...) r mr))]
  [_ (let ([d (strip x)])  (if (self-evaluating? d)  d  (syntax-error x "invalid syntax")))]))
```

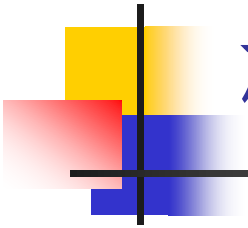


展开器(II)

```
(define (exp-macro p x)
   (let ([m (make-mark)])
      (add-mark m (p (add-mark m x)))))

(define (exp-core p x r mr)
          (p x r mr))

(define (exp-exprs x* r mr)
    (map (lambda (x) (exp x r mr)) x*))
```

核心变换(I)

```
(define (exp-if x r mr)
  (syntax-case x ( )
    [(_ e1 e2 e3)
      `(if ,(exp #'e1 r mr)
      ,(exp #'e2 r mr)
      ,(exp #'e3 r mr))]))
```



核心变换(II)

```
(define (exp-let x r mr)
  (syntax-case x ( )
  [(_ ([var expr]) body)
    (let ([label (make-label)] [new-var (gen-var #'var)])
      `(let ([,new-var ,(exp #'expr r mr)])
         ,(exp (add-subst #'var label #'body)
                 (extend-env label
                     (make-binding 'lexical new-var)
                     r)
                 mr)))]))
```

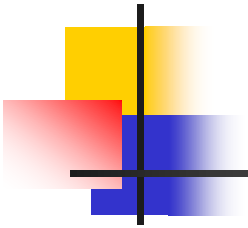
解析和构造句法对象(I)

```
(define (syntax-pair? x)
    (pair? (syntax-object-expr x)))
(define (syntax-car x)
    (extend-wrap
         (syntax-object-wrap x)
    (car (syntax-object-expr x))))
(define (syntax-cdr x)
    (extend-wrap
        (syntax-object-wrap x)
        (cdr (syntax-object-expr x))))
```

解析和构造句法对象(II)

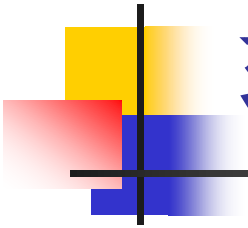
```
(define exp-syntax
    (lambda (x r mr)
        (syntax-case x ( )
            [(_ t) `(quote ,#'t)])))
```

- 模式匹配和重新构造不在本章讨论范围之内，上述只是一些具体实现的基础



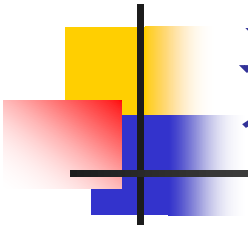
比较标识符

```
(define free-identifier=?
  (lambda (x y)
   (eq? (id-label x) (id-label y))))
(define bound-identifier=?
   (lambda (x y)
(and (eq? (syntax-object-expr x) (syntax-object-expr y))
        (same-marks?
            (wrap-marks (syntax-object-wrap x))
            (wrap-marks (syntax-object-wrap y))))))
```



转换

```
(define datum->syntax
  (lambda (template-id x)
    (make-syntax-object x
      (syntax-object-wrap template-id))))

(define syntax->datum strip)
```



开始展开(I)

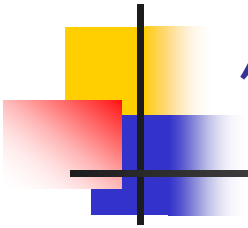
```
(define (expand x)
  (let-values ([(wrap env)
                (initial-wrap-and-env)])
    (exp (make-syntax-object x wrap)
         env env)))
```

开始展开(II)

```
(define initial-wrap-and-env
  (lambda ( )
  (define id-binding*
    `((quote . ,(make-binding 'core exp-quote))
      (if . ,(make-binding 'core exp-if))
      (lambda . ,(make-binding 'core exp-lambda))
      (let . ,(make-binding 'core exp-let))
      (letrec-syntax . ,(make-binding 'core exp-letrec-syntax))
      (identifier? . ,(make-binding 'lexical 'identifier?))
      (free-identifier=? . ,(make-binding 'lexical 'free-identifier=?))
      (bound-identifier=? . ,(make-binding 'lexical 'bound-identifier=?))
      (datum->syntax . ,(make-binding 'lexical 'datum->syntax))
      (syntax->datum . ,(make-binding 'lexical 'syntax->datum))
```

开始展开(III)

```
    (syntax-error . ,(make-binding 'lexical 'syntax-error))
    (syntax-pair? . ,(make-binding 'lexical 'syntax-pair?))
    (syntax-car . ,(make-binding 'lexical 'syntax-car))
    (syntax-cdr . ,(make-binding 'lexical 'syntax-cdr))
    (syntax . ,(make-binding 'core exp-syntax))
    (list . ,(make-binding 'core 'list))))
(let ([label* (map (lambda (x) (make-label)) id-binding*)])
  (values
    `(,@(map (lambda (sym label)
                    (make-subst sym (list top-mark) label))
             (map car id-binding*)
             label*)
        ,top-mark)
     (map cons label* (map cdr id-binding*))))))
```

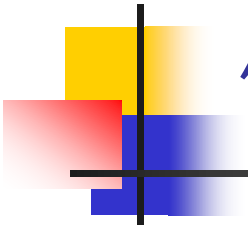


例子(I)

- 依旧是开头的例子：

  (let ([t #t]) (or #f t))
- **or**的宏定义如前所述
- 最开始，展开器接受上述表达式的句法对象。简便起见，假设无初始包装，则起始句法对象应为：

  < (let ([t #t]) (or #f t)) >



例子(II)

- 展开器同时接受初始环境，假定其包含了宏绑定、核心形式及内置过程的绑定，在此均省略。
- 由于**let**出现在初始包装和环境里，它被视为核心形式，其变换器递归地展开为右手边表达式**#t**，并返回**#t**，于是得：

< (or #f t) [t x () -> l1] >

例子(III)

- 环境也被扩展，用新建名字**t.1**把标签映射到**lexical**类型：

  l1 -> lexical(t.1)

- **or**形式被当作宏调用，所以**or**的变换器被调用，生成新的表达式，对该表达式求值在同一环境中进行：

<(<let> ((<t> #f))

(<if> <t> <t> <t m2 [t x () ->l1]>))

m2>

例子(IV)

- 另一个核心**let**表达式，在甄别和解析它的过程中，标记**m2**被推压到子式上：

```
(<let m2> ((<t m2> #f))
    <(<if> <t> <t> <t m2 [t x () -> l1]>)
    m2>)
```

- **let**的变换器递归地展开**#f**，返回**#f**，然后用扩展包装递归地展开表达式体

例子(V)

- 这个包装用标记**m2**把引入的**t**映射到新建标签**l2**上：

<(<if> <t> <t> <t m2 [t x () -> l1]>)

[t x (m2) -> l2]

m2>

- 环境也被扩展，借助新建名**t.2**把标签映射到**lexical**类型的绑定上：

l2 -> lexical(t.2), l1-> lexical(t.1)

例子(VI)

- 得到的表达式被识别为核心形式**if**。在识别和解析的过程中，展开器把外部的替代器和标记压到部件上。上次出现的**t**的包装上已有标记**m2**。该标记与外包装上的标记**m2**消掉，于是**t**没有标记：

```
(<if [t x (m2) -> l2] m2>
    <t [t x (m2) -> l2] m2>
    <t [t x (m2) -> l2] m2>
    <t [t x (m2) -> l2] [t x () -> l1]>)
```

例子(VII)

- 宏**if**的转换器在输入环境里递归地处理**if**子式。首先<t [t x (m2) -> l2] m2>被认做标识符引用，因为该表达式为符号**(t)**。包装里的替换器也类似，故展开器在环境里查找**l2**，发现它映射到词法变量**t.2**。第二个子式相同，但第三个子式中标识符没有**m2**标记，所以第一个替换器不适用，而第二个适用。

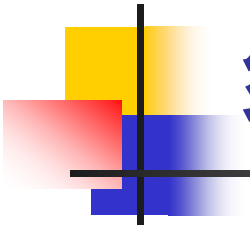
例子(VIII)

- 展开器在环境里查找**l1**，并发现它映射到**t.1**
- 展开快结束时，表达式**if**被重新构造为：

  (if t.2 t.2 t.1)
- 内层**let**被构造为

  (let ([t.2 #f]) (if t.2 t.2 t.1))
- 外层**let**被构造为

  (let ([t.1 #t]) (let ([t.2 #f]) (if t.2 t.2 t.1)))



结论

- 这里展示的简化展开器完整实现了**syntax-case**所需要的基本算法，但并未涉及模式匹配机理问题
- 尽管**syntax-case**展开器相当复杂，但并不有损其美妙，作者认为复杂的软件仍有精妙之处，只要它结构完善并能完成既定的任务

感谢

- 欢迎大家提出问题并讨论！