微通道框架

by Andrew W. Keep
and Leif Andersen

```
(require nanopass/base)                                    package: nanopass
```

nanopass/base 库提供所有的微通道框架。

```
#lang nanopass                                             package: nanopass
```

nanopass 语言提供 nanopass/base 的一切, 以及 racket 语言。 使它适合用作语言。

1 概述

微通道框架提供一种工具，可用于编写由在定义明确的中间语言上运行的几个简单通道组成的编译器。 该组织的目标是简化对每个通道的理解,因为它负责单个任务,并且简化在编译器中任何地方添加新通道的过程。

使用的微通道框架的最完整的公开示例在 `tests/compiler.rkt` 文件中。 这是 Racket 简化子集的编译器实现的开始,在有关编译器实现的课程中使用。

2 定义语言

微通道框架在一组编译器-编写器定义的语言上运行。Racket编程语言的简单变体的语言定义如下所示:

```
(define-language L0
  (terminals
    (variable (x))
    (primitive (pr))
    (datum (d))
    (constant (c)))
  (Expr (e body)
    x
    pr
    c
    'd
    (begin e* ... e)
    (if e0 e1)
    (if e0 e1 e2)
    (lambda (x* ...) body* ... body)
    (let ([x* e*] ...) body* ... body)
    (letrec ([x* e*] ...) body* ... body)
    (e0 e1 ...)))
```

L0 语言由一组终端(以 terminals 形式列出)和 一个非终端 Expr 组成。 这种语言的终端是 variable 、 primitive 、 datum 和 constant。 每个终端都列出一个或多个元变量,可用于在非终端表达式中表示终端。 在这种情况下, variable 可以用 x 表示, primitive 可以用 pr 表示, datum 可以用 d 表示, constant 可以用 c。 微通道框架期望编译器编写器提供与每个终端相对应的谓词。 谓词的名称是从终端的名称派生出来的,通过添加 ? 字符。 对于 L0 语言,需要提供 variable? 、 primitive? 、 datum? 和 constant? 谓词。 可能决定将变量表示为符号,选择一小部分基元,将基准表示为任何Racket值, 并将常量限制为语法不需要引号的那些东西:

```
(define variable?
  (lambda (x)
    (symbol? x)))

(define primitive?
  (lambda (x)
    (memq x '(+ - * / cons car cdr pair? vector make-vector vector-length
              vector-ref vector-set! vector? string make-string
              string-length string-ref string-set! string? void))))

(define datum?
  (lambda (x)
    #t))

(define constant?
  (lambda (x)
    (or (number? x)
        (char? x)
        (string? x))))
```

L0 语言还定义一个非终端 Expr 。 非终端以一个名称 (Expr) 开始,后面是一组元变量 ((e body)) 和一组结果。 每个产生式都遵循三种形式之一。 它要么是一个独立的元变量(在本例中是 x 、 pr 或 c), 要么一个以文字开头的S表达式(在本例中是 'd 、 (begin e* ... e) 、 (if e0 e1) 等),要么是一个不以文字开头的S表达式(在本例中是

(e0 e1 ...))。 除了示例中使用的数字或开头 (*) 后缀外, 问号 (?) 或插入符号 (^) 也可以使用。 这些后缀也可以组合使用。 后缀不包含任何语义值,仅用于允许元变量被多次使用(如 (if e0 e1 e2) 形式)。

最后,为了创建这种语言的元素,可以使用由 define-parser 形式自动生成的解析器来解析一个简单的示例。 默认情况下,还会生成一个非解析器。

```
(define-parser parse-L0 L0)

(parse-L0 '(let ([x 19]) (f x) (g x 4)))

(unparse-L0 (parse-L0 '(let ([x 19]) (f x) (g x 4))))
```

2.1 扩展语言

既然已经有 L0 语言,可以想象指定一种只有两个选择的if(即 (if e0 e1 e2))的语言, 只有带引号的常量,lambda、let和只包含一个表达式的letrec体。 所有旧语言的元素,仅包含想要的更改。这可以很好地工作,但是我们可能想要简单地定义哪些更改。 微通道框架提供一种语法,允许将一种语言定义为对已定义语言的扩展:

```
(define-language L1
  (extends L0)
  (terminals
    (- (constant (c))))
  (Expr (e body)
    (- c
       (if e0 e1)
       (lambda (x* ...) body* ... body)
       (let ([x* e*] ...) body* ... body)
       (letrec ([x* e*] ...) body* ... body))
    (+ (lambda (x* ...) body)
       (let ([x* e*] ...) body)
       (letrec ([x* e*] ...) body))))
```

新定义的语言使用 extends 形式表示 L1 是现有语言 L0 的扩展。 extends 形式更改处理终端和非终端形式的方式。 现在,这些表格应包含一组 - 表示应删除的产生式或终端的表格, 或 + 表示应添加的产生式或终端的表格。也可以添加新的非终端, 只需简单地用单 + 形式添加新的非终端形式,即可添加新的非终端的所有产生式。

2.2 define-language 言形式

define-language 的完整语法如下:

(define-language *language-name clause ...*) syntax

```
                    clause = (extends language-name)
                           | (entry non-terminal-name)
                           | (terminals terminal-clause ...)
                           | (terminals extended-terminal-clause ...)
                           | (non-terminal-name (meta-var ...)
                               production-clause ...)
                           | (non-terminal-name (meta-var ...)
                               extended-production-clause ...)

             terminal-clause = (terminal-name (meta-var ...))
                           | (=> (terminal-name (meta-var ...))
                               prettifier)
                           | (terminal-name (meta-var ...)) => prettifier

    extended-terminal-caluse = (+ terminal-clause ...)
                           | (- terminal-clause ...)

          production-clause = terminal-meta-var
                           | non-terminal-meta-var
                           | production-pattern
                           | (keyword . production-pattern)

  extended-production-clause = (+ production-clause ...)
                           | (- production-clause ...)
```

其中 *clause* 是扩展子句、入口子句、终止子句或非终止子句。

---

extends syntax

extends 子句指出该语言是现有语言的扩展。在语言定义中只能指定一个扩展子句。 extends 子句在其他地方是错误的。

language-name 是已定义语言的名称。

---

entry syntax

`entry` 子句指定哪个非终结符是该语言的起点。 在生成通道时使用此信息来确定通道应首先期望哪个非终端。 这个默认值可以在通道定义中被重写,如 定义通道 所述。 在语言定义中只能指定一个 entry 子句。 `entry` 子句在其他地方是错误的。

`non-terminal-name` 对应于此语言(或其派生的语言)中指定的一个非终端。

---

terminals syntax

`terminals` 子句指定语言使用的一个或多个终端。 例如,在 `L0` 示例语言中,终端子句指定四种终端类型:变量、原语、数据和常量。

`terminal-name` 是终端的名称,存在一个相应的 `terminal-name?` 谓词函数， 来确定一个Racket对象在检查一个通道的输出时是否属于这种类型。 `meta-var` 是元变量的名称,用于在语言和过程定义中引用此终端类型。

`prettifier` 是一种表达式,其计算结果为一个参数的函数, 当在“ pretty”模式下调用语言unparser来生成漂亮的S表达式表示时,将使用该参数。

最后的形式是它上面的形式的语法糖。 如果省略 `prettifier` ,则在unparser运行时不会在终端上进行任何处理。

当一种语言派生于一种基本语言时,使用 `extended-terminal-clause` 。 `+` 形式表示应该添加到新语言中的终端。 `-` 形式表示在生成新语言时应该从旧语言列表中删除的终端。 在 terminals 子句中未提及的终端将被复制为新语言,并且保持不变。

请注意 ,当前从终端添加和删除 `meta-var` 要求删除终端类型并重新添加。 可以使用如下所示的 terminal 子句在同一步骤中完成此操作:

```
(terminals
  (- (variable (x)))
  (+ (variable (x y))))
```

`terminals` 子句在其他地方是错误的。

2.2.1 非终端子句

non-terminals 子句使用一种语言指定有效的产生式。每个非终端符都有一个名称,一组元变量和一组产生式。

`non-terminal-name` 是命名非终端的标识符, `meta-var` 是在语言中引用该非终端符， 并在通道定义时使用的元变量名, 而 `production-clause` 有以下形式之一:

`terminal-meta-var` 是一个终端元变量,它是此非终端的独立生产式。

`non-terminal-meta-var` 是一个非终端的元变量, 它指示指定的非终端允许的任何形式也被该非终端允许。

`keyword` 是在解析S表达式表示形式,语言输入模式或语言输出模板时必须完全匹配的标识符。

`production-pattern` 是表示语言模式的S表达式,具有以下形式。

```
          production-pattern = meta-variable
                             | (maybe meta-variable)
                             | (production-pattern-sequence ...)
                             | (production-pattern-sequence ...
                                 . production-pattern)

  production-pattern-sequence = production-pattern
                              | ... ; literal ...
```

`meta-variable` 是任何终端符或非终端的元变量,扩展为任意数字, 后跟 `*` 、 `?` 或 `^` 字符的任意组合, 例如,如果元变量是 `e` ,则为 `e1` 、 `e*` 、 `e?` 和 `e4*?` 都是有效的元变量表达式。

`(maybe meta-variable)` 表示产生式中的元素要么是元变量的类型, 要么是元变量的类型或底部(用 `#f` 表示)。

因此,Racket 的语言形式如 `let` 可以表示为如下语言产生式:

```
(let ([x* e*] ...) body* ... body)
```

其中 `let` 是 `keyword`, `x*` 是一个元变量,表示变量列表, `e*` & `body*` 是一个元变量,表示表达式列表, `body` 是一个元变量,表示单个表达式。 类似于named-let形式的内容也可以表示为:

```
(let (maybe x) ([x* e*] ...) body* ... body)
```

尽管这与普通的命名let形式略有不同,但是非命名形式将需要一个显式的 `#f` 来指示未指定名称。

当语言从基本语言扩展时,请使用 `extended-production-clause` 。 `+` 形式表示应该添加到新语言的非终端符的非终端符产生式。 `-` 形式表示在生成新语言时,应该从此非终端旧语言的产生式列表中删除的非终端产生式。 除去此非终端的旧的语言非终端制作。非终端条款中未提及的产生式将复制到新语言的非终端中,并且保持不变。 如果新语言的非终端程序删除所有产生式,则非终端程序在新语言中被删除。 相反,可以通过命名新的非终端并使用 `+` 来添加新的非终端。

2.3 define-language 的产生式

`define-language` 形式产生两个用户可见的对象:

- 语言定义,绑定到指定的 `language-name`;
- 一个非解析器(名为 `unparse-language-name`) , 可用于将基于记录的表示解析回s表达式;

在将 `language-name` 指定为新语言定义的基础时, 在通道的定义中以及为了生成解析器时,将使用语言定义。

(**define-parser** *parser-name language-name*) syntax

将名称 *parser-name* 绑定到 *language-name* 语言的自动生成解析器。

(**language->s-expression** *language-name*) syntax

基于 *language-name* 生成一个S表达式。

对于扩展语言,这将生成完整语言的S表达式。

对于 `L1`:

```
(language->s-expression L1)
```

将返回:

```
(define-language L1
  (terminals
    (variable (x))
    (primitive (pr))
    (datum (d)))
  (Expr (e body)
    (letrec ([x* e*] ...) body)
    (let ([x* e*] ...) body)
    (lambda (x* ...) body)
    x
    pr
    'd
    (begin e* ... e)
    (if e0 e1 e2)
    (e0 e1 ...)))
```

3 定义通道

通道用于指定在使用 `define-language` 定义的语言上的转换。 在深入定义通道的细节之前,看看从 `L0` 转换到 `L1` 的简单通道。 此通道将需要:

- 去掉一个选择的 `if` ;
- 在语言中引用常量;
- 在 `lambda` 、 `let` 和 `letrec` 形式的主体中使 `begin` 成为显式形式。

可以定义一个名为 `make-explicit` 的通道,使所有这些形式显式。

```
(define-pass make-explicit : L0 (ir) -> L1 ()
  (definitions)
  (Expr : Expr (ir) -> Expr ()
    [,x x]
    [,pr pr]
    [,c `',c]
    [',d  `',d]
    [(begin ,[e*] ... ,[e]) `(begin ,e* ... ,e)]
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]
    [(if ,[e0] ,[e1] ,[e2]) `(if ,e0 ,e1 ,e2)]
    [(lambda (,x* ...) ,[body*] ... ,[body])
     `(lambda (,x* ...) (begin ,body* ... ,body))]
    [(let ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(let ([,x* ,e*] ...) (begin ,body* ... ,body))]
    [(letrec ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(letrec ([,x* ,e*] ...) (begin ,body* ... ,body))]
    [(,[e0] ,[e1] ...) `(,e0 ,e1 ...)])
  (Expr ir))
```

通道定义以名称(在本例中为 `make-explicit`) 和签名开始。 该签名以一个输入语言说明符(在本例中为 `L0`) 以及一个形式列表开始。 在这种情况下,只有一个正式的 `ir` 输入语言术语。 签名的第二部分有一个输出语言说明符(在本例中为 `L1`)以及一个额外返回值列表(在本例中为空)。

在名称和签名之后,通道可以定义一组额外的 `definitions` (在本例中为空)、 一组处理器(在本例中为 `(Expr : Expr (ir) -> Expr () ---)`) 和一个体表达式(在本例中为 `(Expr ir)`)。 与通道类似,处理器从名称(在本例中是 `Expr`) 和签名开始。 签名的第一部分是非终端说明符(在本例中是 `Expr`) 以及形式列表(在本例中是 `ir`)。 这里的 `Expr` 对应于输入语言 `L0` 中定义的表达式。 输出包括一个非终端输出说明符(在本例中也是 `Expr`)和一个额外的返回表达式列表。

签名后,有一组子句。每个子句由一个输入模式、一个可选子句和一个或多个表达式组成, 这些表达式指定一个或多个基于签名的返回值。输入模式源自输入语言中指定的S表达式。 模式中的每个变量都用 unquote (`,`)表示。 当取消引号后跟一个S表达式时(例如 `(begin ,[e*] ... ,[e])` 中的 `,[e*]` 和 `,[e]` 中的情况) ,它表示 cata-morphism。 cata-morphism 对所提到的子表单执行自动递归。

再次查看示例通道,可能会注意到 `(definitions)` 形式是空的。 当它为空时,不需要包含它。 体表达式 `(Expr ir)` 只是调用与输入语言的入口点对应的处理器。 这可以通过 `define-pass` 自动生成,因此也可以从定义中省略它。 最后,可能会注意到有几个子句只是简单地匹配输入模式, 并生成一个完全匹配的输出模式(对嵌套的 `Expr` 表单的拼接态取模)。 因为输入和输出语言已经定义,所以 `define-pass` 宏也可以自动生成这些子句。再试一次:

```
(define-pass make-explicit : L0 (ir) -> L1 ()
  (Expr : Expr (ir) -> Expr ()
    [,c `',c]
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]
    [(lambda (,x* ...) ,[body*] ... ,[body])
     `(lambda (,x* ...) (begin ,body* ... ,body))]
    [(let ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(let ([,x* ,e*] ...) (begin ,body* ... ,body))]
    [(letrec ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(letrec ([,x* ,e*] ...) (begin ,body* ... ,body))]))
```

尽管这两种通道都是有效的方式,但这种是更简单。

3.1 define-pass 形式

```
(define-pass name : lang-specifier (formal ...)                    syntax
                  -> lang-specifier (extra-return-val ...)
  definitions-clause
  processor ...
  body-expr)

  language-specifier = language-name
                     | (language-name non-terminal-name)
                     | *

  definitions-clause =
                     | (definitions definition-clause ...)

           processor = (processor-name
                        : non-terminal-spec (formal ...)
                        -> non-terminal-spec (extra-return-val ...)
                        definition-clause
                        processor-clause ...)

    non-terminal-spec = non-terminal-name
                      | *

     processor-clause = [pattern body-expr ...+]
                      | [pattern (guard expr ...+) body-expr ..+]
                      | [else body-expr ...+]
                      | expr
```

name 是通道的名称。

formal 是表示形式参数的标识符。

language-name 是与定义的语言相对应的标识符。

non-terminal-name 是该语言的名称。

extra-return-val 是 Racket 表达式。

expr 是一个 Racket 表达式,可以包含 *quasiquote-expr*。

definition-clause 是 Racket 定义。

body-expr 是 Racket 表达式。

quasiquote-expr 是一个 Racket quasiquote 表达式, 它与处理器中指定的输出非终端形式相对应。

non-terminal-spec 决定处理器的行为。 如果处理器的输入 *non-terminal-spec* 是 `*`, 那么处理器的主体是 Racket 表达式而不是微通道表达式。 如果处理器的输出 *non-terminal-spec* 是 `*`, 那么 `quasiquote` 将是 Racket 的 `quasiquote`,而不是生成输出语言。 此外, *extra-return-val* 中的第一个值将是处理器返回的第一个默认值。

否则, *non-terminal-spec* 描述输入和输出语言的非终端。

`define-pass` 宏重新绑定 Racket 准引用,用于在输出语言中构造记录。 类似于处理器子句中的模式,可以将Ququote表达式任意嵌套，以创建嵌套在其他语言记录中的语言记录。 当文字表达式(例如符号、数字或布尔表达式)是记录的一部分之一时,不必将其取消引用。 任何未加引号的文字将直接按原样放入记录中。 只要表达式的输出与记录中期望的类型相对应,则无引号的表达式可以包含任意表达式。

创建准引用表达式时,重要的是要认识到,尽管输入和输出看起来都像任意S表达式,但实际上它们并不是任意S表达式。 例如, `L1` 中的 `let` 的S表达式,会看到:

```
(let ([x* e*] ...) body)
```

这将创建一个包含三个部分的记录:

- 变量 (`x*`) 的列表;
- 表达式 (`e*`) 的列表;
- 一个表达式 (`body`)。

如果这只是一个S表达式,可能会想到将输出记录构造为:

```
; binding*: is ([x0 e0] ... [xn en])
(let ([binding* (f ---)])
  `(let (,binding* ...) ,body))
```

然而,这是行不通的,因为基础设施不知道如何将 `binding*` 分解为其各个部分。 相反,在将 `binding*` 列表包含在准引号表达式中之前,需要对它进行解构:

```
; binding*: is ([x0 e0] ... [xn en])
(let ([binding* (f ---)])
  (let ([x* (map car binding*)]
        [e* (map cadr binding*)])
    `(let ([,x* ,e*] ...) ,body)))
```

3.2 准引用在哪里绑定,它绑定在什么位置?

微通道框架将处理器子句的主体内的准引用重新绑定到由终端类型指定的输出语言的非终端。 例如,如果一个通道具有带有以下签名的处理器:

```
(define-pass my-pass L0 (ir) -> L1 ()
  ---
  (Expr : Expr (ir) -> Expr ()
    ---)
  (Stmt : Stmt (ir) -> Stmt ()
    ---))
```

... 然后,准引号将绑定到 `Expr` 处理器中的L1 Expr非终端和Stmt处理器中的L1 Stmt。

在准引用表达式中,取消引用会将准引用重新绑定到适当的非终端。 例如,假设我们的语言有以下产生式:

```
(define-language L1
  (terminals ---)
  (Expr (e)
    ---)
  (Stmt (stmt)
    ---
    (define x e)))
```

... 那么上面的Stmt处理器中的准引用可能具有以下形式:

```
`(define ,x ,code-to-build-an-Expr)
```

在 `'code-to-build-an-expr` 章节中,准引号将反弹到L1 `Expr` 非终端。 当然,你可能想要这样写代码:

```
(let ([e ,code-to-build-an-expr])
  `(define ,x ,e))
```

因为 `code-to-build-an-expr` 现在处于L1 `Stmt` 非终端的上下文中。 可以使用微通道的上下文形式把我们置于 `Expr` 的上下文中。

(**in-context** *nonterminal form*) syntax

将准引号绑定到一个表单, 该表单在对 *form* 求值期间构造当前输出语言的 *nonterminal* 表单的元素。

因此,可以这样写:

```
(let ([e (in-context Expr ,code-to-build-an-expr)])
  `(define ,x ,e))
```

... ， 并且构建 `Expr` 的代码，现在将具有正确的准引用。

3.3 如何在处理器之外(甚至通道)构建语言形式?

基本上有两种方法可以在处理器(或过程)之外构建语言形式。 第一种方法是使用语言解析器,它对于源语言的S表达式版本中的大量固定代码很有用。 可以使用上述的 `define-parser` 定义语言解析器。

例如,

```
(define-parser parse-L L)
```

... 将为语言L构建解析器,并将其绑定到函数parse-L。

第二种更常见的方法是使用with-output-language形式。 这种形式允许您将处理器中使用的准引号s表达式语法与在其他地方生成的表达式混合使用。 with-output-language 形式有两种形式。

(with-output-language *Language form0 ... form*) syntax
(with-output-language (*Language Nonterminal*) *form0 ... form*)

将 `in-context` (如上所述)绑定到与指定语言关联的表单。

在第二个示例中, `quasiquote` 也绑定到构造指定项的元素的宏。

本质上,第二种形式只是以下形式的缩写:

```
(with-output-language Language (in-context Nonterminal) form0 ... form)
```

因此,给定语言:

```
(define-language L
  (terminals
    (symbol (x))
    ---)
  (Expr (e)
    ---
    (let ([x* e*] ...) e)))
```

可以编写函数:

```
(define buld-let
  (lambda (x* e* body)
    (with-output-language (L Expr)
      `(let ([,x* ,e*] ...) ,body))))
```

或者

```
(with-output-language (L Expr)
  (define buld-let
    (lambda (x* e* body)
      `(let ([,x* ,e*] ...) ,body))))
```

... 因为包装多个定义时, `with-output-language` 将扩展为开始拼接形式。

4 在通道之外匹配语言形式

除了 `define-pass` 形式外,还可以使用 `nanopass-case` 形式来匹配语言术语。 这在创建功能抽象(例如基于匹配语言形式提出问题的谓词)时很有用。 例如,假设为 `L8` 语言编写一个 `lambda?` 谓词,如下所示:

```
(define (lambda? e)
    (nanopass-case (L8 Expr) e
      [(lambda (,x* ...) ,abody) #t]
      [else #f]))
```

`nanopass-case` 形式的语法如下:

(nanopass-case (*lang-name nonterminal-name*) *expr* syntax
matching-clause ...)

matching-clause = [*pattern guard-clause expr ... expr*]
| [*pattern expr ... expr*]
| [*else expr ... expr*]

给定一个语言/非终止对,一个表达式以及使用先前定义的模式语言的一系列模式/结果对, 请评估第一个匹配模式的右侧。

从本质上, `nanopass-case` 提供一种更简洁的替代方法来定义单独的 `define-pass` 形式。

事实上, `nanopass-case` 扩展成 `define-pass` 。 具体地说,前面的 `lambda?` 示例可以重写为

```
(define-pass lambda? : (L8 Expr) (e) -> * (bool)
  (Expr : Expr (e) -> * (bool)
    [(lambda (,x* ...) ,abody) #t]
    [else #f])
```

```
  (Expr e))
```

4.1 催化同态

Cata-morphisms在模式中定义为未引用的S表达式。cata-morphism具有以下语法:

```
cata-morphism = ,[identifier ...]
              | ,[identifier expr ... -> identifer ...]
              | ,[func-expr : identifier expr ... -> identifier ...]
              | ,[func-expr : -> identifier ...]
```

expr 是一个 Racket 表达式, *func-expr* 是一个函数的表达式， *identifier* 是一个标识符, 当它位于 `->` 的左侧时, 它将绑定到输入子表达式;当它在右侧时,它将绑定到函数的返回值。

当未指定 *func-expr* 时, `define-pass` 尝试查找或自动生成适当的处理器。 当 *func-expr* 是处理器的名称时, 将检查输入非终端、输出非终端以及输入表达式和返回值的数量,以确保它们匹配。

如果省略输入表达式,则第一个输入将是子表达式,并将按名称从包含处理器的形式中获取附加值。

4.2 自动生成的子句

当处理器的输入语言非终端和输出语言非终端中存在匹配形式时,可以在处理器内自动生成子句。 例如,可以消除示例中的 `x` 、 `pr` 和 `'d` 子句, 因为这些形式既存在于输入语言的 `Expr` 非终端中, 也存在于输出语言的 `Expr` 非终端中。 自动生成的子句还可以递归地处理子表单。 这就是为什么 `(begin ,e* ... ,e)` 、 `(if ,e0 ,e1 ,e2)` 和 `(,e0 ,e1 ...)` 子句可以省略。

自动生成的子句还可以处理具有多个非终结符的语法。对于自动生成子句中的每个子术语,选择一个处理器。 对于自动生成子句中的每个子术语,将选择一个处理器。根据给定子终端的输入非终端和输出非终端选择处理器。 除了输入和输出非终端外,处理器所需的任何其他自变量都必须可用, 并且处理器必须返回自动生成的子句所需的任何其他返回值。如果找不到合适的处理器, 则将在可能的情况下自动生成一个新的处理器(这将在下一节中进行描述),否则将在编译时引发异常。

当处理器中包含 `else` 子句时,将不会自动生成任何子句,因为预计else子句将处理所有其他情况。

4.3 自动生成的处理器

当催化形态,自动生成的子句或自动生成的通道主体表达式需要处理一个子项, 但不存在可以处理该子项的处理器时,微通道框架可能会创建一个新的处理器。 仅当期望输入语言中的单个参数和输出语言中的单个返回值,并且输出语言非终端包含与 所有输入语言非终端匹配的生成时,才能创建处理器。这意味着输出语言非终端可能包含额外的形式, 但不能遗漏任何输入项。

该功能当前是在框架中实现的,但是使用该功能可能会遇到麻烦。通常,在两种情况下可以自动生成不需要的子句。

- 子句中的嵌套模式与所有可能的子句匹配,但是由于嵌套了该模式,因此框架无法确定这一点。 在这种情况下,自动生成的子句会拖累系统,但是将永远不会调用自动生成的处理器, 并且实际上无法访问生成的代码。
- cata-morphism或自动生成的子句提供单个输入表达式,并且需要单个输出表达式, 但是不存在可以满足此匹配条件的处理器。这可能会导致创建一组影子处理器, 并且通过通道可能会产生不希望的结果。

第一项是可解决的问题,尽管我们尚未投入开发时间来解决它。 第二个问题更难解决,也是该功能仍处于试验阶段的原因。