神奈川大学理学部　情報科学科　馬谷研究室

Racket 言語のマクロ記述用ライブラリの比較

山内 雄登 201903038

1 はじめに

Racket とは, Scheme から派生したプログラミング言語である.Racket には, Macro-By-Example (MBE) という堅牢なマクロを簡単に書けるツールが存在する [3]. 例として, syntax-rules や syntax-case 等が挙げられる.

しかし, MBE は構文パターンが弱く, プログラマは検証やエラー報告の作業をガードやトランスフォーマーに移行せざるを得ない. さらに, ガード式は節全体を受け入れるか拒否するか, そして拒否はなぜガードが失敗したかについての情報なしで行われる. 最後に, MBE は重要な構文を幅広く記述するための語彙を欠いている. 本調査ではこのような問題を解決する syntax-parse について調査する.

2 syntax-parse

syntax-parse はドメイン固有の言語を用いて構文解析, 検証, エラー報告を行う. このシステムは, MBE に対して 3 つの重要な改善点を備えている.

- 構文パターンを表現する言語であり, マッチング可能な構文クラスで注釈されたパターン変数を含む.
- 構文パターンを抽象化した新しい構文クラスを定義する機能.
- 進捗状況を追跡し, 失敗をランク付けして報告するマッチングアルゴリズムと, エラー情報を伝える失敗の概念.

さらに, ガード式はサイドコンディションに置き換えられ, 拒否のメッセージが表示される.

構文クラスの追加により, 宣言的な仕様とハンドコーディングされたチェックを規律正しく織り交ぜることができる.

3 構文の検証

let を例として syntax-parse の設計を説明する.

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let ([var rhs]...) body)
     #'((lambda (var ...) body ...) rhs ...)]))
```

var とラベル付けされた項はすべて識別子でなければならないという制約を加える. 同様に, rhs と body は式であることを示すアノテーションが付けられている. 最後の制約である識別子が一意であることは`#:fail-when` 節を使用してサイドコンディションとして表現されている. 以下は修正されたマクロである.

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let([var:id rhs:expr] ...) body:expr)
     #:fail-when (check-duplicate #'(var ...))
                 "duplicate variable name"
     #'((lambda (var ...) body) rhs ...)]))
```

構文クラスの注釈はパターン変数の一部ではないため, テンプレートに表示されないことに注意する.

サイドコンディションはガード式と異なり, 生成される失敗には失敗の理由を記述する情報が含まれる.

この時点で, let マクロはその構文を適切に検証している. 以下は, 誤用の例である.

```
> (let ([x 1] [x 2]) (h x))
let: duplicate variable name in : x
> (let ([(x y) (f 7)]) (g x y))
let: expected identifier in : (x y)
```

しかし, いくつかの誤用については, let はまだ良いエラーメッセージを表示しない.

```
> (let (x 5) (add1 x))
let: bad syntax
```

この let マクロは, 一般的なエラーメッセージでこの誤用を拒否する. より良いエラーメッセージを得

るには, マクロ作成者が syntax-parse に追加情報を提供する必要がある.

4 syntax-class の定義

構文クラスは syntax-parse のエラー報告メカニズムの基礎を形成している. バインディングペアのための構文クラスを定義することで,syntax-parse に新しいクラスのエラーを説明するための語彙を与える. バインディングペアの構文は, このように構文クラスとして定義される.

```
(define-syntax-class binding
  #:description "binding pair"
  (pattern (var:id rhs:expr)))
```

パターン変数 var と rhs はメインパターンから構文クラスに移動したため, それらのバインディングがメインパターンで利用できるように, 構文クラスの属性としてエクスポートする必要がある. バインディング注釈付きのパターン変数 b は, 属性の名前と組み合わされて, ネストされた属性 b.var と b.rhs を形成する.

パターンだけでなく, 構文クラスはサイドコンディションを含むことができる.

```
(define-syntax distinct-bindings
  #:description "sequence of binding pairs"
  (pattern (b:binding ...)
           "duplicate variable name"
  #:with (var ...) #'(b.var ...)
  #:with (rhs ...) #'(b.rhs ...)))
```

with 節は, パターンと計算された用語をマッチさせる。ここでは, var と rhs を distinct-bindings の属性としてバインドするために使用している. デフォルトでは, 構文クラスはそのパターンのパターン変数のみを属性としてエクスポートし, ネストした属性はエクスポートしない.distinct-bindings の var および rhs 属性は省略の深さが 1 であるため, マクロのパターン内の省略記号内に bs が存在しなくても, マクロのテンプレート内の省略記号内で bs.var と bs.rhs を使用することができる.

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let bs:distinct-bindings body:expr)
     #'((lambda (bs.var ...) body)
        bs.rhs ...)]))
```

binding と distinct-bindings の構文が指定されたので,syntax-parse はこれらを使用して let の追加の誤用に対する良いエラーメッセージを生成することができる.

```
> (let (x 5) (add1 x))
let: expected binding pair in: x
> (let 17)
let: expected sequence of binding pair in: 17
```

5 エラーの報告

しかし, 今のところこのマクロは Racket の let が提供する機能の半分しか実装していない. そこで,"named-let"という形式を追加する.

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let loop:id bs:distinct-bindings body:expr)
     #'(letrec ([loop (lambda (bs.var ...) body)])
         (loop bs.rhs ...))]
    [(let bs:distinct-bindings body:expr)
     #'((lambda (bs.var ...) body) bs.rhs...)]))
```

distinct-bindings 構文クラスを再利用することができるため,named-let の追加は簡単にできる.

複数の構文解析節や構文クラス定義の複数のバリエーションなど, 複数の代替案がある場合, それらは順番に試される. 代替案が失敗すると, syntax-parse は失敗を記録し, 次の代替案にバックトラックを行う. 代替案が試されると, syntax-parse は失敗のリストを蓄積し, それぞれの失敗にはマッチングの進捗状況の測定値が含まれる. マッチングプロセス全体が失敗した場合, 最も進展した試みが選ばれ, シンタックスエラーが説明される.

6 まとめ

syntax-parse は，エラー報告型のバックトラック・アルゴリズムと表現力豊かなパターン・ランゲージに基づいて，明確かつ堅牢なマクロを書くためのドメイン特化型言語である.

参考文献

[1] Racket Documentation/Syntax: Meta-Programming Helpers/1.1 Introduction. https://docs.racket-lang.org/syntax/stxparse-intro.html

[2] Culpepper, R., Felleisen, M., Fortifying macros, ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming, Pages 235–246. https://doi.org/10.1145/1863543.1863577

[3] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In Symposium on Principles of Programming Languages, pages 77–84, 1987.