A Survey Language

```
form Box1HouseOwning {
   hasSoldHouse: "Sold a house in 2010?" boolean
   hasBoughtHouse: "Bought a house in 2010?" boolean
   hasMaintLoan: "Did you enter a loan?" boolean
   if (hasSoldHouse) {
     sellPrice: "Price sold for:" money
     debt: "Private debts:" money
     valueResidue: "Residue:" money (sellPrice - debt)
   }
}
```

A Survey Language

```
(form Box1HouseOwning
      [hasSoldHouse "Sold a house in 2010?" boolean]
      [hasBoughtHouse "Bought a house in 2010?" boolean]
      [hasMaintLoan "Did you enter a loan?" boolean]
      (when hasSoldHouse
        [sellPrice "Price sold for:" money]
        [debt "Private debts:" money]
        [valueResidue "Residue:" money (- sellPrice debt)]))
```

Version 0: Syntactic Abstraction

define a pattern-based macro

definition

```
(define-simple-macro (form name clause ...)
  (begin
    (define name (make-gui 'name))
    (form-clause name #t clause) ...
    (send name start)))
```

---

use

```
(form Box1
      [hasSoldHouse "Sold a house?" boolean]
      [hasBoughtHouse "Bought a house?" boolean]
      (when hasSoldHouse
        [price "Selling price:" money]
        [debt "Private debts:" money]))
```

Version 0: Syntactic Abstraction

definition

pattern

```
(define-simple-macro (form name clause ...)
  (begin
    (define name (make-gui 'name))
    (form-clause name #t clause) ...
    (send name start)))
```

---

use

```
(form Box1
      [hasSoldHouse "Sold a house?" boolean]
      [hasBoughtHouse "Bought a house?" boolean]
      (when hasSoldHouse
        [price "Selling price:" money]
        [debt "Private debts:" money]))
```

Version 0: Syntactic Abstraction

definition

```
(define-simple-macro (form name clause ...)
  (begin
    (define name (make-gui 'name))
    (form-clause name #t clause) ...
    (send name start)))
```

template

---

use

```
(form Box1
      [hasSoldHouse "Sold a house?" boolean]
      [hasBoughtHouse "Bought a house?" boolean]
      (when hasSoldHouse
        [price "Selling price:" money]
        [debt "Private debts:" money]))
```

Version 0: Syntactic Abstraction

definition

```
(define-simple-macro (form name clause ...)
  (begin
    (define name (make-gui 'name))
    (form-clause name #t clause) ...
    (send name start)))
```

---

use

```
(form Box1
      [hasSoldHouse "Sold a house?" boolean]
      [hasBoughtHouse "Bought a house?" boolean]
      (when hasSoldHouse
        [price "Selling price:" money]
        [debt "Private debts:" money]))
```

Version 0: Syntactic Abstraction

definition

```
(define-simple-macro (form name clause ...)
  (begin
    (define name (make-gui 'name))
    (form-clause name #t clause) ...
    (send name start)))
```

---

use

```
(form Box1
      [hasSoldHouse "Sold a house?" boolean]
      [hasBoughtHouse "Bought a house?" boolean]
      (when hasSoldHouse
        [price "Selling price:" money]
        [debt "Private debts:" money]))
```

Version 0: Syntactic Abstraction

definition

```
(define-syntax (form-clause stx)
  (syntax-parse stx
    #:literals (when)
    [(_ form-name guard (when guard2 nested ...))
     #'(begin
         (form-clause form-name (and guard guard2) nested)
         ...)]
    [(_ form-name guard [id question type])
     ....]
    [(_ form-name guard [id question type expr])
     ....]))
```

---

use

```
(form-clause Box1 #t
             [hasSoldHouse "Sold a house?" boolean])
```

basic clause

```
(form-clause Box1 #t
             [hasBoughtHouse "Bought a house?" boolean])
(form-clause Box1 #t
             (when hasSoldHouse
               [price "Selling price:" money]
               [debt "Private debts:" money]))
```

compound with guard

Version 0: Syntactic Abstraction

definition

define a macro

match multiple patterns

```
(define-syntax (form-clause stx)
  (syntax-parse
    #:literals (when)
    [(_ form-name guard (when guard2 nested ...))
     #'(begin
         (form-clause form-name (and guard guard2) nested)
         ...)]
    [(_ form-name guard [id question type])
     ....]
    [(_ form-name guard [id question type expr])
     ....]))
```

---

use

```
(form-clause Box1 #t
             [hasSoldHouse "Sold a house?" boolean])
(form-clause Box1 #t
             [hasBoughtHouse "Bought a house?" boolean])
(form-clause Box1 #t
             (when hasSoldHouse
               [price "Selling price:" money]
               [debt "Private debts:" money]))
```

Version 0: Syntactic Abstraction

definition

```
(define-syntax (fo use stx)
  (syntax-parse st
    #:literals (when)
    [(_ form-name guard (when guard2 nested ...))
     #'(begin
         (form-clause form-name (and guard guard2) nested)
         ...)]
    [(_ form-name guard [id question type])
     ....]
    [(_ form-name guard [id question type expr])
     ....]))
```

literals

---

use

```
(form-clause Box1 #t
             [hasSoldHouse "Sold a house?" boolean])
(form-clause Box1 #t
             [hasBoughtHouse "Bought a house?" boolean])
(form-clause Box1 #t
             (when hasSoldHouse
               [price "Selling price:" money]
               [debt "Private debts:" money]))
```

Version 0: Syntactic Abstraction

definition

```
(define-syntax (form-clause stx)
  (syntax-parse stx
    #:literals (when)
    [(_ form-name guard (when guard2 nested ...))
     #'(begin
         (form-clause form-name (and guard guard2) nested)
         ...)]
    [(_ form-name guard [id question type])
     ....]
    [(_ form-name guard [id question type expr])
     ....]))
```

---

use

```
(form-clause Box1 #t
             [hasSoldHouse "Sold a house?" boolean])
(form-clause Box1 #t
             [hasBoughtHouse "Bought a house?" boolean])
(form-clause Box1 #t
             (when hasSoldHouse
               [price "Selling price:" money]
               [debt "Private debts:" money]))
```

Version 1: Syntactic Extension

```
#lang racket
(provide form
         (rename-out [boolean-widget boolean]
                     [boolean-money money])
         .....)
.....

(define-simple-macro (form name clause ...)
  (begin
    (define name (make-gui 'name))
    (form-clause name #t clause) ...
    (send name start)))

.....
```

not exported, works anyway: macro scope

form.rkt

```
#lang racket
(require "form1.rkt")

(form Box1 .....)
```

survey.rkt

Version 2: Module Language

```
#lang racket
(provide form
         .....

         when .....

         #%module-begin)
.....
```

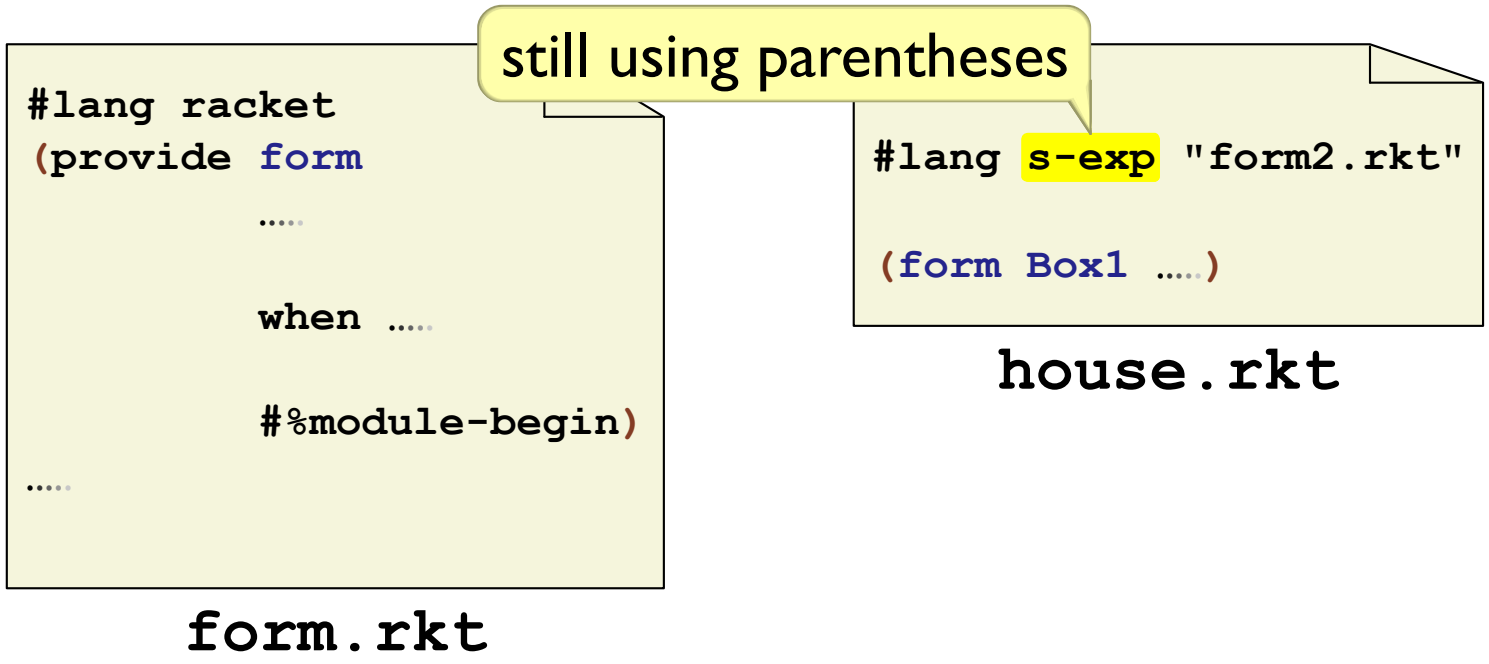
form.rkt

specify language

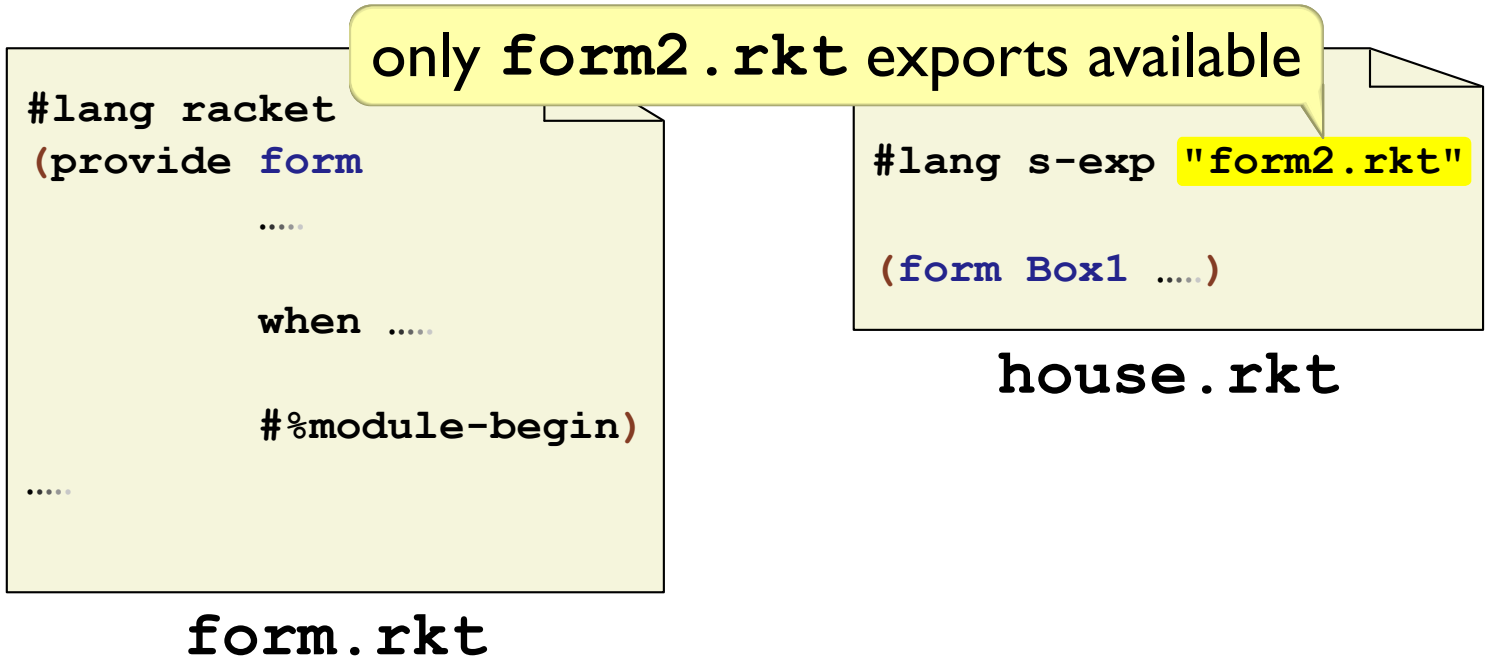
```
#lang s-exp "form2.rkt"

(form Box1 .....)
```

house.rkt

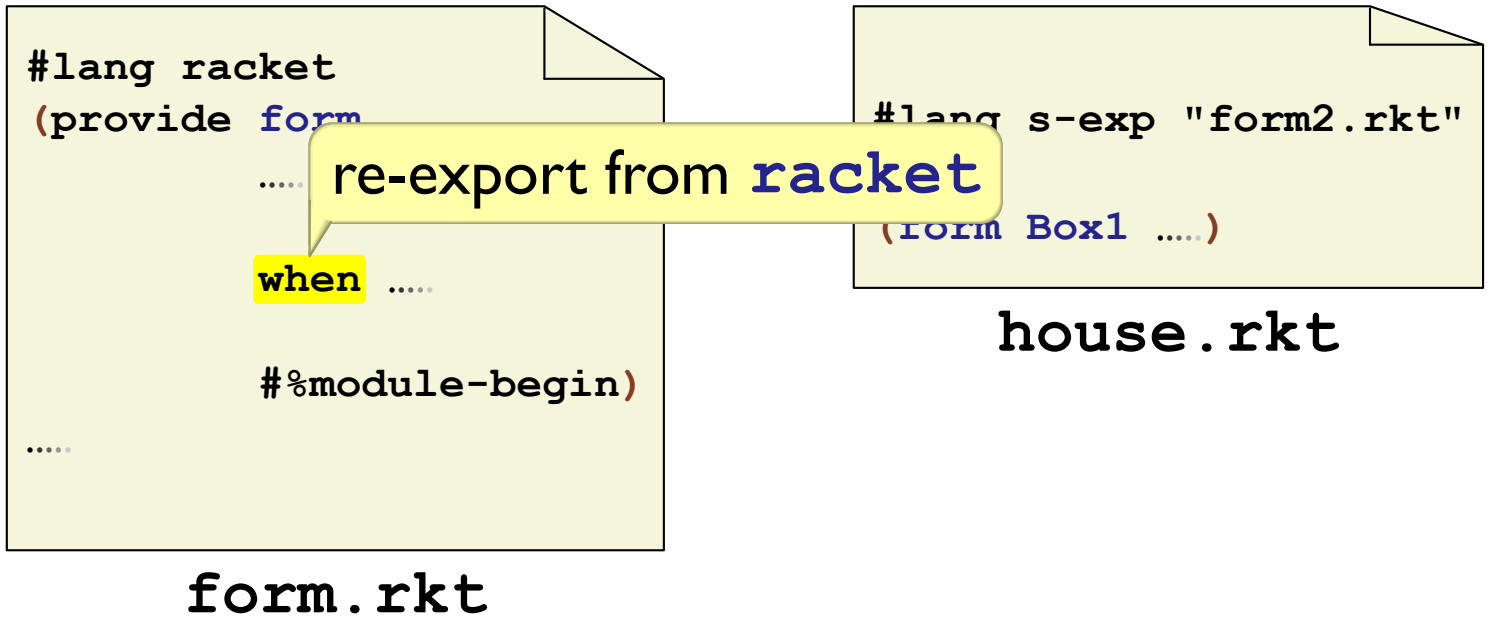
Version 2: Module Language



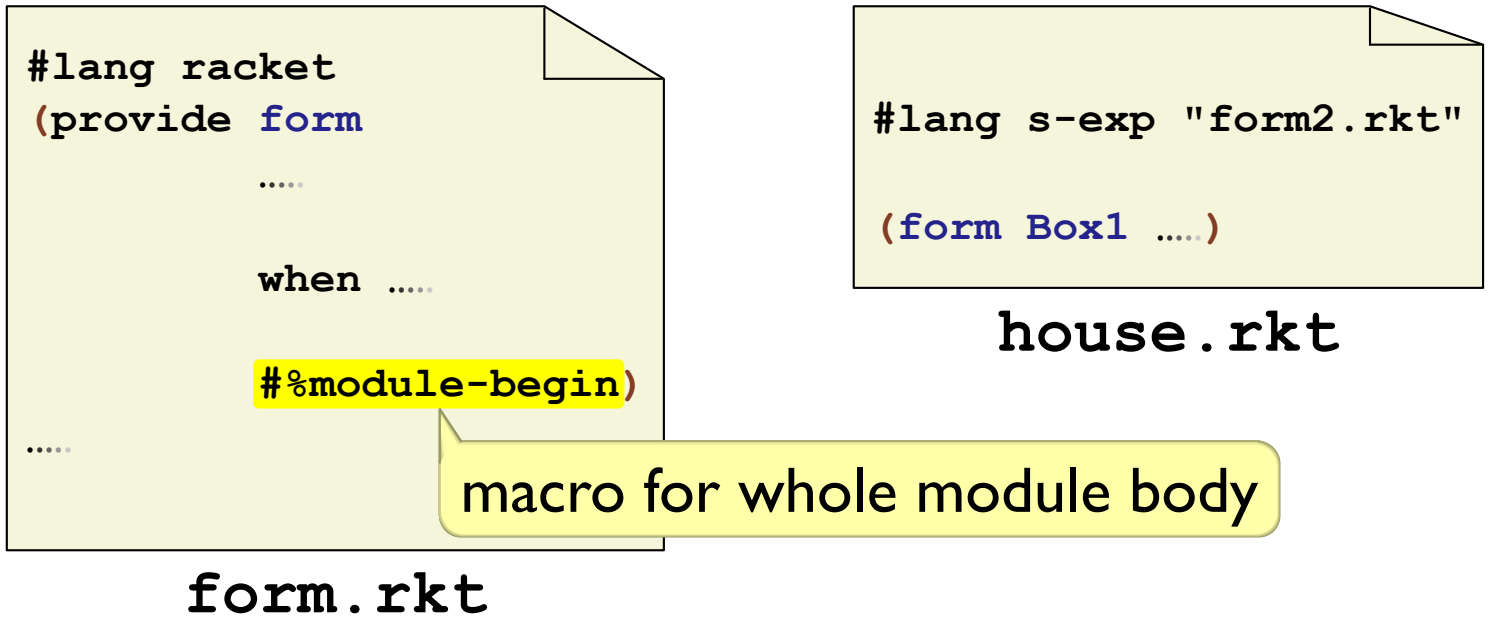
Version 2: Module Language



Version 2: Module Language



Version 2: Module Language



Version 3: Avoiding Duplication

```
(form Pick
      [n1 "A number:" number]
      (when (prime? n1)
        [n2 "Another number:" number]
        (when (prime? n2)
          [n3 "A number:" number])))
```

expands to

```
.....
(form-clause* Pick #t
              n1 "A number:" number #f)
(form-clause* Pick (and #t
                        (prime? n1))
              n2 "A number:" number #f)
(form-clause* Pick (and (and #t
                             (prime? n1))
                        (prime? n2))
              n3 "A number:" number #f)
```

Version 3: Avoiding Duplication

```
(define-syntax (form-clause stx)
  (syntax-parse stx
    #:literals (when)
    [(_ name guard (when expr nested ...))
     #'(begin
         (define new-guard
           (lambda () (and (guard) expr)))
         (form-clause name new-guard nested)
         ...)]
    ....))
```

Version 3: Avoiding Duplication

```
(form Pick
      [n1 "A number:" number]
      (when (prime? n1)
        [n2 "Another number:" number]
        (when (prime? n2)
          [n3 "A number:" number])))
```

expands to

```
.....
(form-clause* Pick (lambda () #t)
              n1 "A number:" number #f)
(define new-guard1
  (lambda () (and ((lambda () #t)) (prime? n1))))
(form-clause* Pick new-guard1
              n2 "A number:" number #f)
(define new-guard2
  (lambda () (and (new-guard1) (prime? n2))))
(form-clause* Pick new-guard2
              n3 "A number:" number #f)
```

Version 4: Improve Syntax Checking

```
(define-simple-macro (form name clause ...)
  (begin
    (define name (make-gui 'name))
    ....))
```

Version 4: Improve Syntax Checking

```
(define-syntax (form stx)
  (syntax-parse stx
    [(_ name clause ...)
     #'(begin
         (define name (make-gui 'name))
         ....)]))
```

Version 4: Improve Syntax Checking

```
(define-syntax form
  (lambda (stx)
    (syntax-parse stx
      [(_ name clause ...)
       #'(begin
           (define name (make-gui 'name))
           ....)])))
```

Version 4: Improve Syntax Checking

```
(define-syntax form
  (lambda (stx)
    (syntax-parse stx
      [(_ name clause ...)
       (unless (identifier? #'name)
         (raise-syntax-error ....))
       #'(begin
           (define name (make-gui 'name))
           ....)])))
```

Version 5: Types

```
(form
 .....
 [residue "Value residue:" money hasSoldHouse]
 .....)
```

~~*boolean shows up in money field*~~

syntax error: `hasSoldHouse' does not have type `money'

Version 5: Types

start compile-time code

```
(begin-for-syntax
  (define (typed id type)
    (lambda (stx) ….  #`(has-type #,id #,type)  ….)))

(define-syntax check-type
  (lambda (stx)
    ….  (local-expand #'expr 'expression (list #'has-type))  ….))
```

```
(define-syntax form-clause
  (lambda (stx)
    ….
    [(_ form-name guard-proc [id question type expr])
     #'(form-clause* form-name guard-proc id question type
                     (lambda () (check-type expr type)))]
    ….))

(define-syntax form-clause*
  (lambda (stx)
    …. #'(begin
           (define val-id undefined)
           (define-syntax id (typed #'val-id #'type))
           ….)))
```

Version 5: Types

a compile-time function declaration

```
(begin-for-syntax
  (define (typed id type)
    (lambda (stx) ….  #`(has-type #,id #,type)  ….)))

(define-syntax check-type
  (lambda (stx)
    ….  (local-expand #'expr 'expression (list #'has-type))  ….))
```

```
(define-syntax form-clause
  (lambda (stx)
    ….
    [(_ form-name guard-proc [id question type expr])
     #'(form-clause* form-name guard-proc id question type
                     (lambda () (check-type expr type)))]
    ….))

(define-syntax form-clause*
  (lambda (stx)
    …. #'(begin
           (define val-id undefined)
           (define-syntax id (typed #'val-id #'type))
           ….)))
```

Version 5: Types

```
(begin-for-syntax
  (define (typed id type)
    (lambda (stx) ….  #`(has-type #,id #,type)  ….)))

(define-syntax check-type
  (lambda (stx)
    ….  (local-expand #'expr 'expression (list #'has-type))  ….))
```

```
(define-syntax form-clause
  (lambda (stx)
    ….
    [(_ form-name guard-proc [id question type expr])
     #'(form-clause* form-name guard-proc id question type
                     (lambda () (check-type expr type)))]
    ….))

(define-syntax form-clause*
  (lambda (stx)
    …. #'(begin
           (define val-id undefined)
           (define-syntax id (typed #'val-id #'type))
           ….)))
```

bind *id* to a compile-time function

Version 5: Types

```
(begin-for-syntax
  (define (typed id type)
    (lambda (stx) ….  #`(has-type #,id #,type)  ….)))

(define-syntax check-type
  (lambda (stx)
    ….  (local-expand #'expr 'expression (list #'has-type))  ….))
```

```
(define-syntax form-clause
  (lambda (stx)
    ….
    [(_ form-name guard-proc [id question type expr])
     #'(form-clause* form-name guard-proc id question type
                     (lambda () (check-type expr type)))]
    ….))
```

check type of *expr* while expanding

```
(define-syntax form-clause*
  (lambda (stx)
    …. #'(begin
           (define val-id undefined)
           (define-syntax id (typed #'val-id #'type))
           ….)))
```

Version 5: Types

```
(begin-for-syntax
  (define (typed id type)
    (lambda (stx) ..... #`(has-type #,id #,type) .....)))

(define-syntax check-type
  (lambda (stx)
    ..... (local-expand #'expr 'expression (list #'has-type)) .....))

(define-syntax form-clause
  (lambda (stx)
    .....
    [(_ form-name guard-proc [id question type expr])
     #'(form-clause* form-name guard-proc id question type
                     (lambda () (check-type expr type)))]
    .....))

(define-syntax form-clause*
  (lambda (stx)
    ..... #'(begin
              (define val-id undefined)
              (define-syntax id (typed #'val-id #'type))
              .....)))
```

force nested expansion

Version 5: Types

```
(begin-for-syntax
  (define (typed id type)
    (lambda (stx) ....  #`(has-type #,id #,type)  ....)))

(define-syntax check-type
  (lambda (stx)
    ....  (local-expand #'expr 'expression (list #'has-type))  ....))
```

```
(define-syntax form-clause
  (lambda (stx)
    ....
    [(_ form-name guard-proc [id question type expr])
     #'(form-clause* form-name guard-proc id question type
                     (lambda () (check-type expr type)))]
    ....))
```

```
(define-syntax form-clause*
  (lambda (stx)
    ....  #'(begin
              (define val-id undefined)
              (define-syntax id (typed #'val-id #'type))
              ....)))
```

= compile time
= run time
= bridge

Version 6: New Language

import character-level parser...

```
#lang reader "reader.rkt"

form Box1HouseOwning {
   hasSoldHouse: "Sold a house in 2010?" boolean
   hasBoughtHouse: "Bought a house in 2010?" boolean
   hasMaintLoan: "Did you enter a loan?" boolean
   if (hasSoldHouse) {
     sellPrice: "Price sold for:" money
     debt: "Private debts:" money
     valueResidue: "Residue:" money (sellPrice - debt)
   }
}
```

house.svy

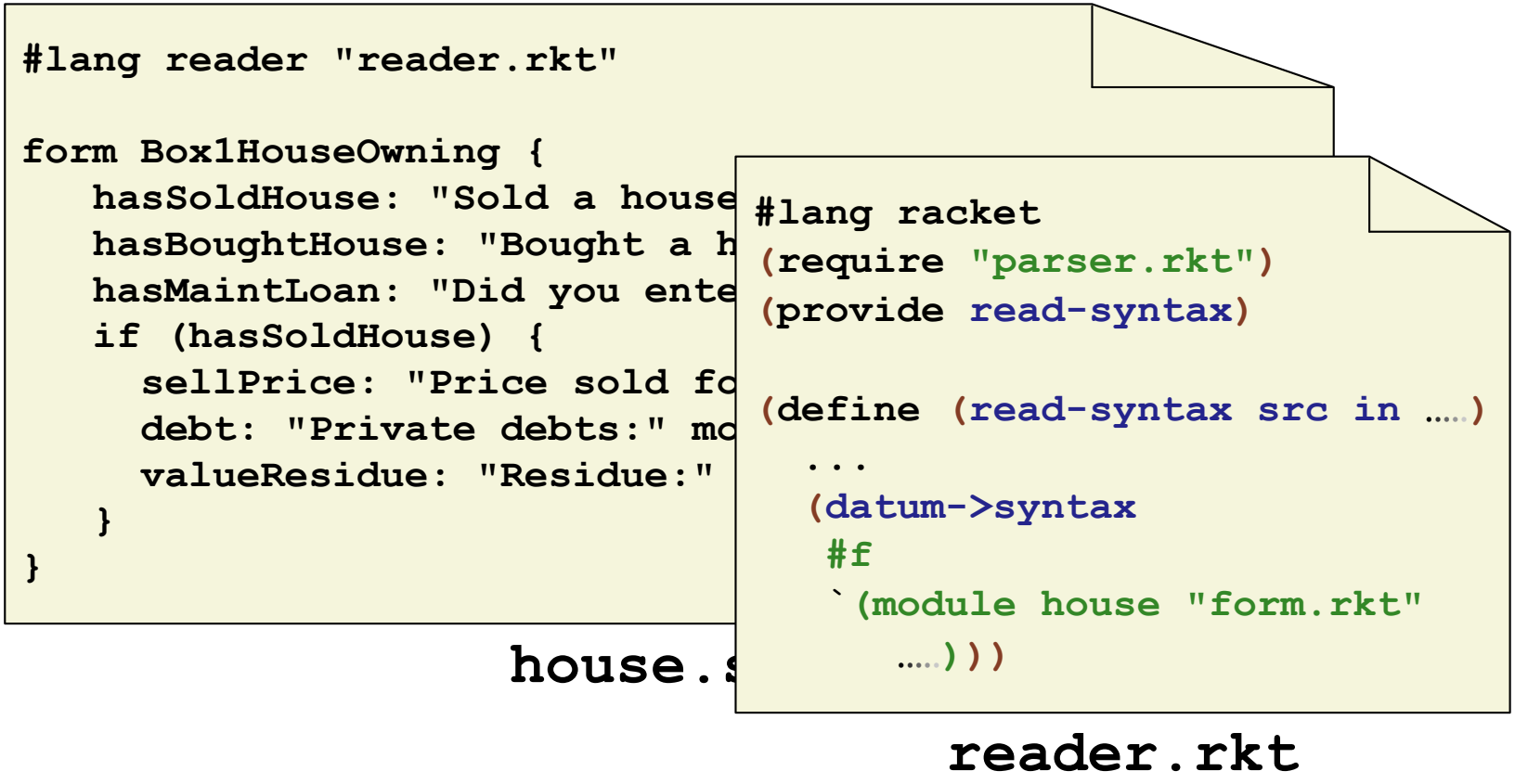
Version 6: New Language

parses into a module that imports **`form.rkt`**

```
#lang reader "reader.rkt"

form Box1HouseOwning {
   hasSoldHouse: "Sold a house in 2010?" boolean
   hasBoughtHouse: "Bought a house in 2010?" boolean
   hasMaintLoan: "Did you enter a loan?" boolean
   if (hasSoldHouse) {
     sellPrice: "Price sold for:" money
     debt: "Private debts:" money
     valueResidue: "Residue:" money (sellPrice - debt)
   }
}
```

`house.svy`

Version 6: New Language



Version 7: Environment Support

installed with **`raco pkg`**

```
#lang survey-dsl

form Box1HouseOwning {
   hasSoldHouse: "Sold a house in 2010?" boolean
   hasBoughtHouse: "Bought a house in 2010?" boolean
   hasMaintLoan: "Did you enter a loan?" boolean
   if (hasSoldHouse) {
     sellPrice: "Price sold for:" money
     debt: "Private debts:" money
     valueResidue: "Residue:" money (sellPrice - debt)
   }
}
```

`house.svy`

Line Counts

Macros:	150
Operations:	40
Parser:	160
Colorer:	50
GUI:	100
Total:	500