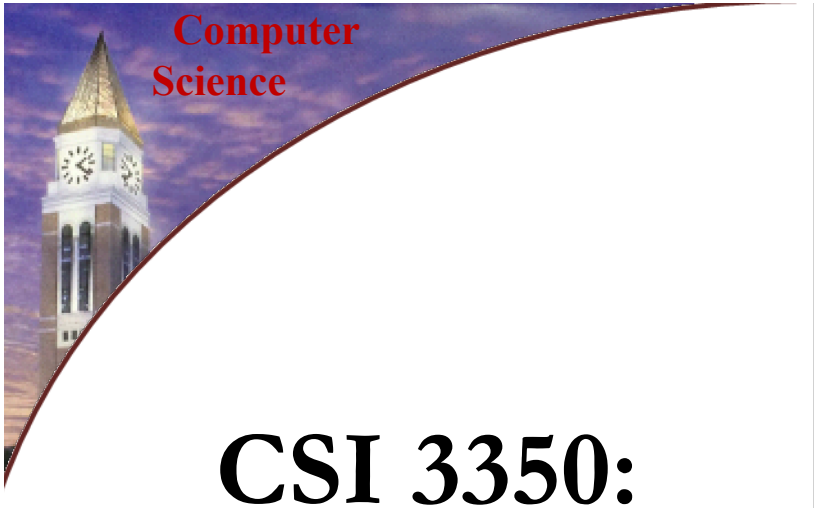
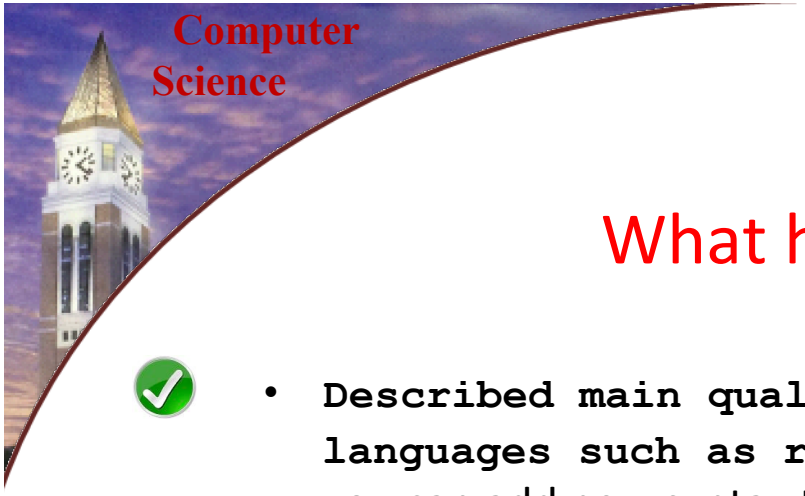


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University

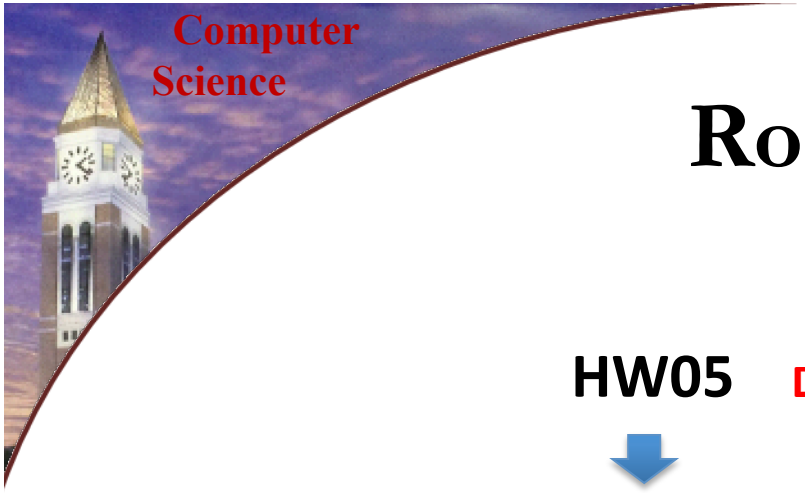


What have we covered so far ?

- **Described main quality criteria for high level programming languages such as readability, writability etc.** (for example you can add new syntax to the language to facilitate the readability of your program using *define-syntax-rule, define-syntax etc.*)
- **Described syntax of fundamental program components** (hw06 *loop, block structure, scoping mechanism, etc.*,)
- **Discussed fundamental concepts of operational semantics** (hw06, coding the **`value-of`** function)

yet to cover

- Describe parameter passing and access to non-locals (hw07 – soon!)
- **Described data types and type systems** (*hw06, grammar, hw05, define-datatype*)
- **Apply major features of functional programming languages** (hw01~hw04, map, foldl, high order functions, lambda etc.)
- **Described activation records** (Sep 30 lecture notes, slides 43 ~ 66)



Road Ahead -

HW05 Due: Nov 12

↓

HW06 Out Nov 13 (today), Due: Nov 24

↓

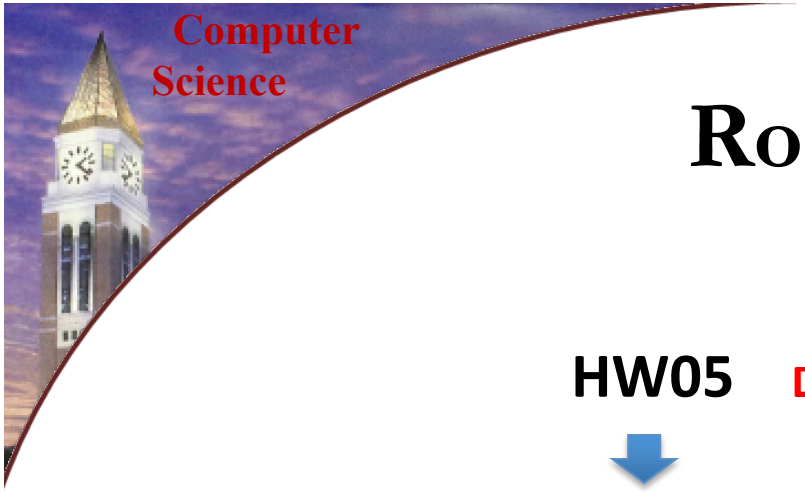
Exam02 ← **Nov 27**

↓

HW07

↓

Final Exam : 7pm ~10pm : Dec 09, 2019



Road Ahead -

HW05 **Due: Nov 12**

↓

HW06 **Out Nov 13 (today), Due: Nov 24**

↓

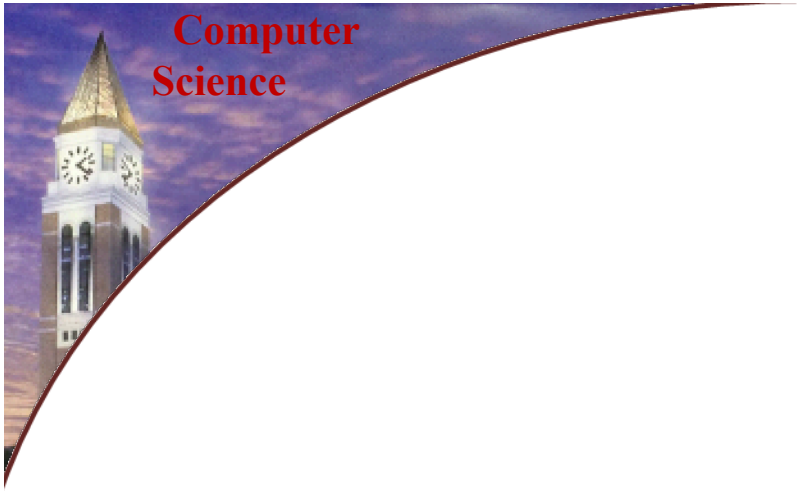
Exam02 ← **Nov 27**

↓

HW07 adding procedure definition and procedure call (parameter passing) into the language implemented in hw06

↓

Final Exam : 7pm ~10pm : Dec 09, 2019



closed book, closed notes, no electronic devices, but **1 page cheat-sheet allowed**

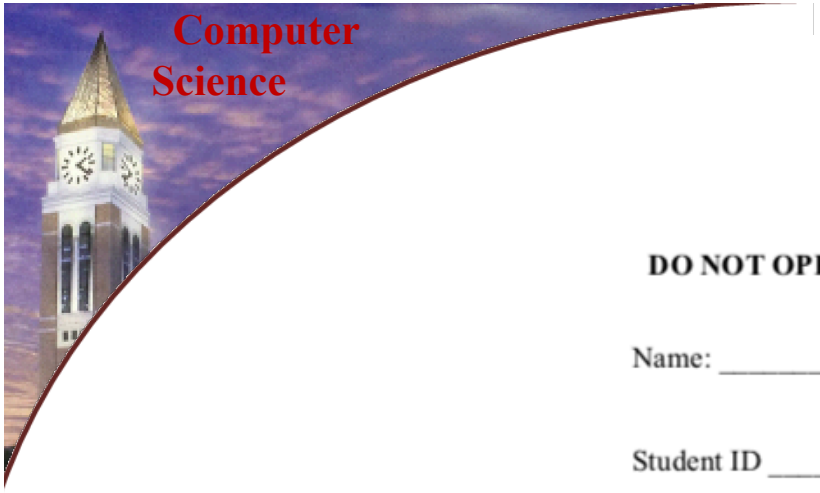
Exam 02

Time: 5:30pm ~ 7:17pm Wednesday, Nov 27th

Location: MSC 185

Coverage: HW05~06

```
Extend, and Modify the language implemented
        in HW06 with new features.
```



CSI 3350
Fall 2019
Exam 2

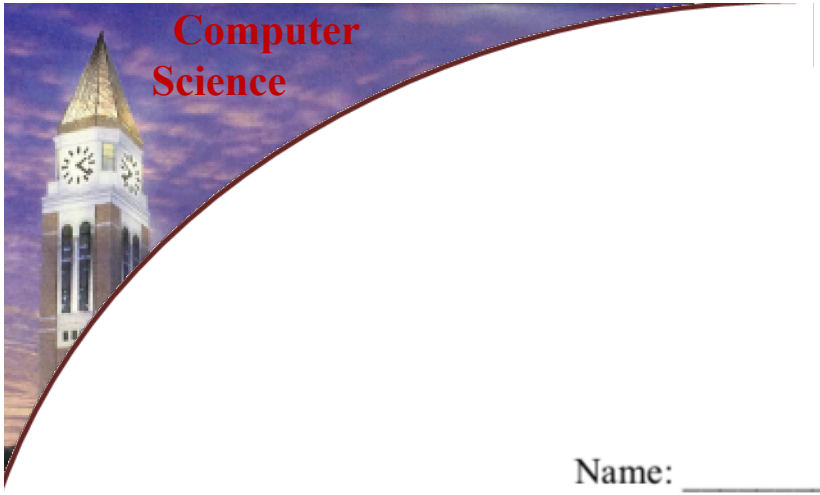
DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: _____

Student ID _____

- This exam is paper-based, closed book, no electronic devices, no headphones.
- Time limit: **110 minutes.**
- You are provided the following in a separate brochure:
  1. `hw06-solution.rkt`: the solution code for homework06
  2. `hw06-env-values.rkt`: which contains related data type definitions.
- Partial credit may be given for partially correct solutions.
- Use correct Scheme syntax; obvious syntax errors will cause deduction of points.
- Indentation is important to us for "clarity" reasons.
- You can use one (1) page (No bigger than 8.5 x 11 inch size, two (2) sides, no less than 9pt font) of notes, aka "cheat sheet". Handwriting is okay. No photo-reduction is permitted. **These notes are to be handed in at the end of the test**. Have your name in the top right corner.
- You can use helping procedures whenever you like.
- Any questions please raise your hand and ask the exam proctor(s).

Good luck!



CSI 3350
Fall 2019

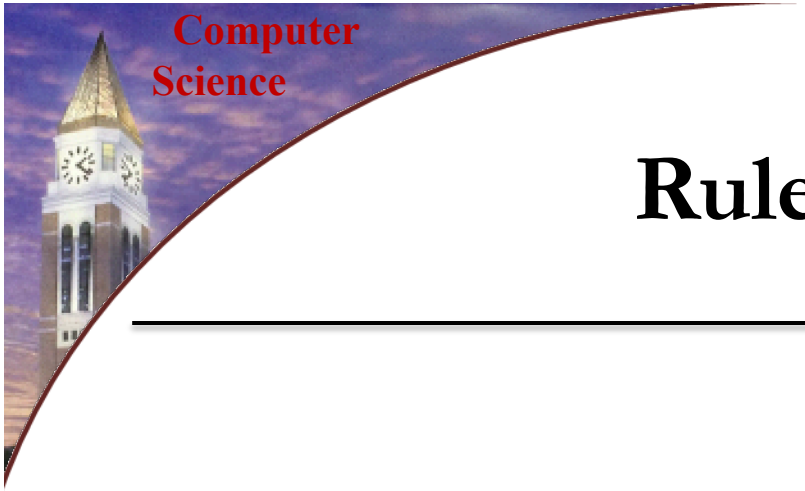
Exam 1 Make-up Questions

Name: _____

Student ID _____

- It is completely up to you if you would like to answer the following questions as a make-up for **exam 01**.
- Should you decide to complete them, you would receive up to 20 extra points, depending on the correctness of your answer. The summation of the earned **extra points** with your current score of exam 01 will be the **updated score of exam 01**.
- Any questions please raise your hand and ask the exam proctor(s).

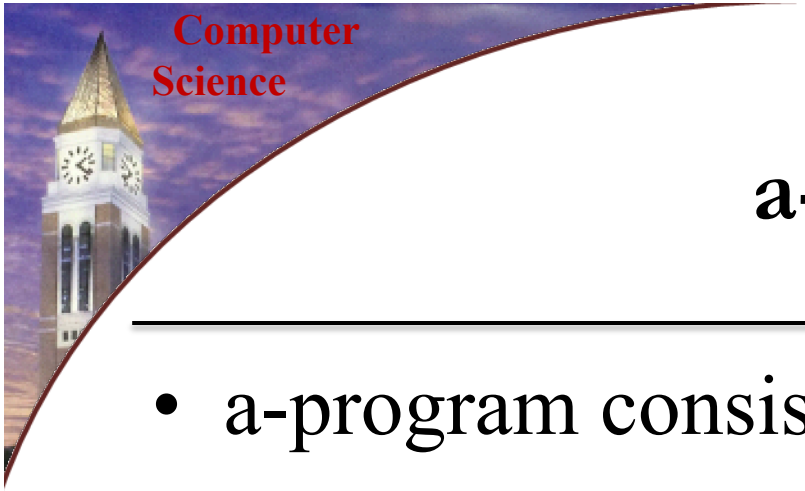
Thank you!



Rule Of Thumb

Every expression will return a value!

For a program consisting of multiple expressions, the **last** expression's value will be the value of the overall expression!



a-program

- a-program consists of one or more expressions

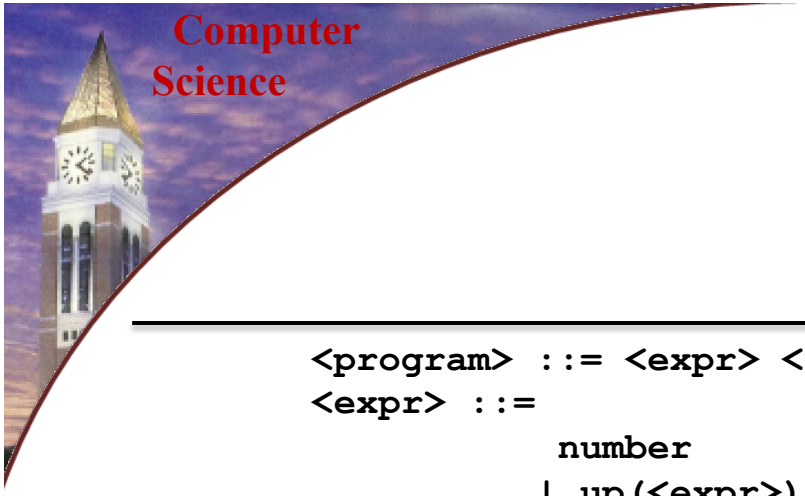
```
(run "42")
```

```
(run "42
      up(3)
      #comment
      down(4)
      111")
```

HW06

(Problem 3)

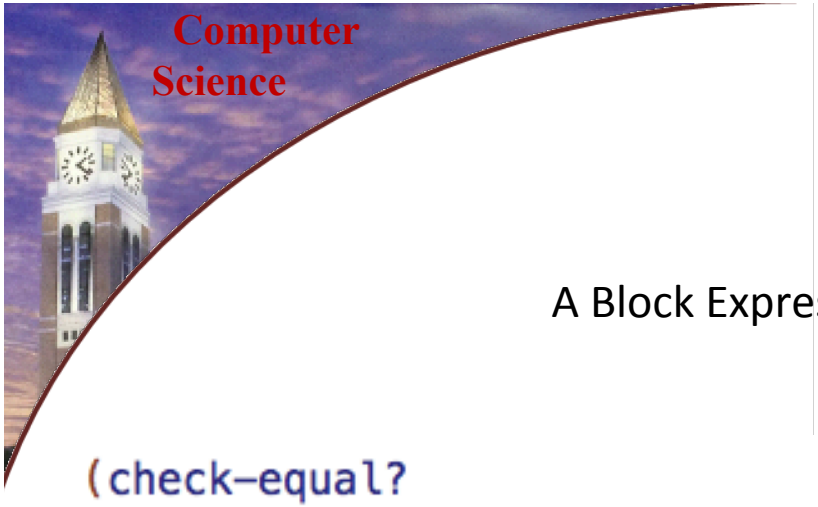
```
<program> ::=
              <expr> <expr>* a-program
<expr> ::=
            number                "num-expr"
          | up(<expr>)            "up-expr"
          | down(<expr>)          "down-expr"
          | left(<expr>)          "left-expr"
          | right(<expr>)         "right-expr"

          | (<expr> <expr>)       "point-expr"
          | + <expr> <expr>       "add-expr"
          | origin? (<expr>)      "origin-expr"
          | if (<expr>)
             then <expr>
             else <expr>           "if-expr"

          | move (<expr> <expr> <expr>*)   "move-expr"
```



HW06

(Problem 4)

```
<program> ::= <expr> <expr>*                                  "a-program"
<expr> ::=
            number                                             "num-expr"
          | up(<expr>)                                        "up-expr"
          | down(<expr>)                                      "down-expr"
          | left(<expr>)                                      "left-expr"
          | right(<expr>)                                     "right-expr"
          | (<expr> <expr>)                                   "point-expr"
          | + <expr> <expr>                                   "add-expr"
          | origin? (<expr>)                                  "origin-expr"
          | if (<expr>)then <expr> else <expr>                  "if-expr"
          | move (<expr> <expr> <expr>*)                        "move-expr"
          | identifier                                        "iden-expr"
          | {<var-expr>* <expr>*}                             "block-expr"

<var-expr> ::= val identifier = <expr>                         "val"
             | final val identifier = <expr>                   "final-val"
```

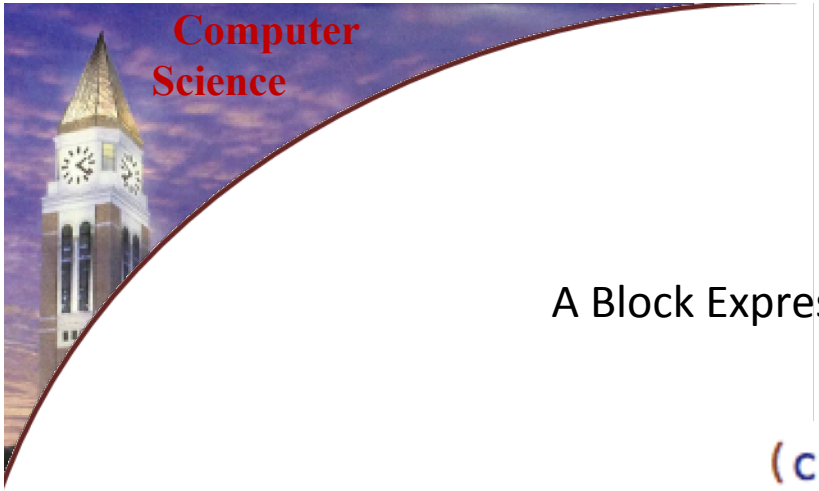


A Block Expression Example (no nested blocks)

```
(check-equal?
 (run "{
          val x = up(3)
          val y = down(4)
          val z = + x y
          z
       }")
 (step-val (down-step 1))
"you should be able to make use of previous variable definitions"
 )
```

Declaration List

→ Value of the overall `block expression`

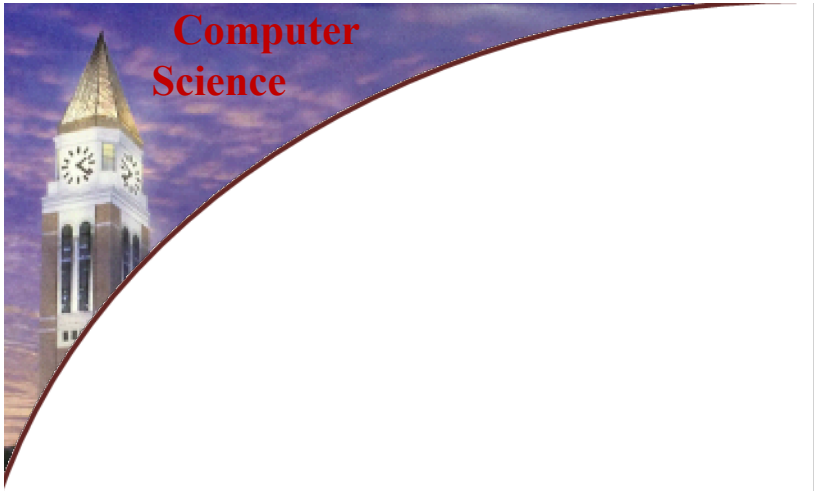


A Block Expression Example (with nested blocks)

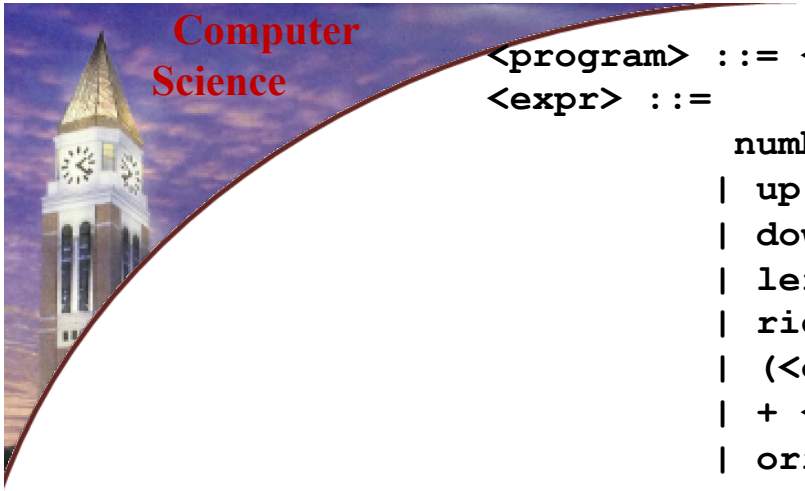
```
(check-equal?
 (run
  "{
     val x = 42
     {
      val y = 23
      x
     }
  }")
 (num-val 42)
 )
```

Nested Block Expression

Outer Block Expression

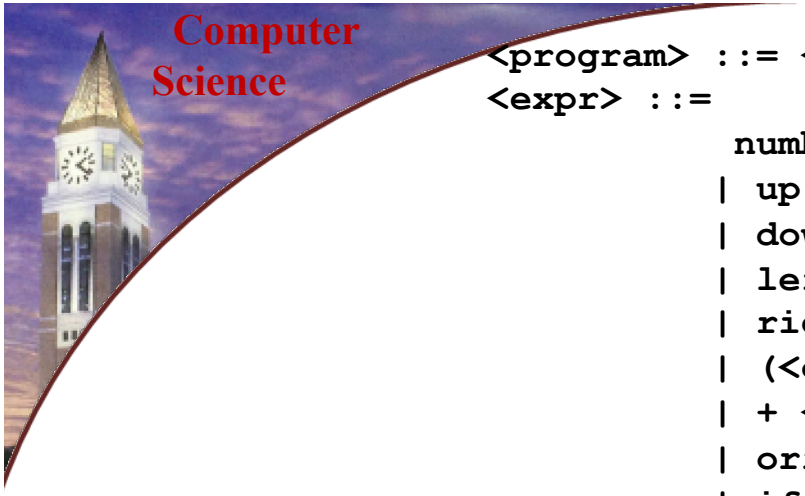



```
(run "{
        val x = 42
        x
        val y = 33


      }"))
```




```
<program> ::= <expr> <expr>*                          "a-program"
<expr> ::=
               number                                  "num-expr"
             | up(<expr>)                              "up-expr"
             | down(<expr>)                            "down-expr"
             | left(<expr>)                            "left-expr"
             | right(<expr>)                           "right-expr"
             | (<expr> <expr>)                         "point-expr"
             | + <expr> <expr>                         "add-expr"
             | origin? (<expr>)                        "origin-expr"
             | if (<expr>)then <expr> else <expr>        "if-expr"
             | move (<expr> <expr> <expr>*)              "move-expr"
             | identifier                              "iden-expr"
             | {<var-expr>* <expr>*}                   "block-expr"

<var-expr> ::= val identifier = <expr>                  "val"
                 | final val identifier = <expr>         "final-val"
```

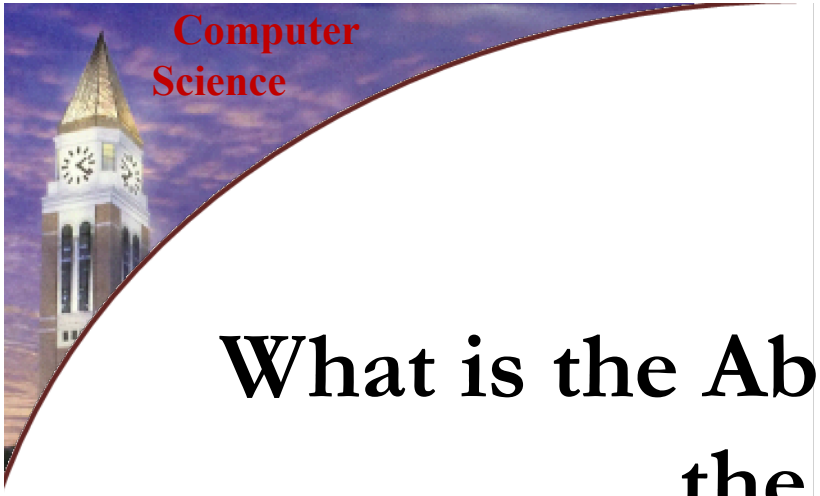
```
(run "{
        val x = 42
        x
        val y = 33


      }"))
```




```
<program> ::= <expr> <expr>*                          "a-program"
<expr> ::=
               number                                  "num-expr"
             | up(<expr>)                              "up-expr"
             | down(<expr>)                            "down-expr"
             | left(<expr>)                            "left-expr"
             | right(<expr>)                           "right-expr"
             | (<expr> <expr>)                         "point-expr"
             | + <expr> <expr>                         "add-expr"
             | origin? (<expr>)                        "origin-expr"
             | if (<expr>)then <expr> else <expr>        "if-expr"
             | move (<expr> <expr> <expr>*)              "move-expr"
             | identifier                              "iden-expr"
             | {<var-expr>* <expr>*}                   "block-expr"

<var-expr> ::= val identifier = <expr>                  "val"
             | final val identifier = <expr>            "final-val"
```

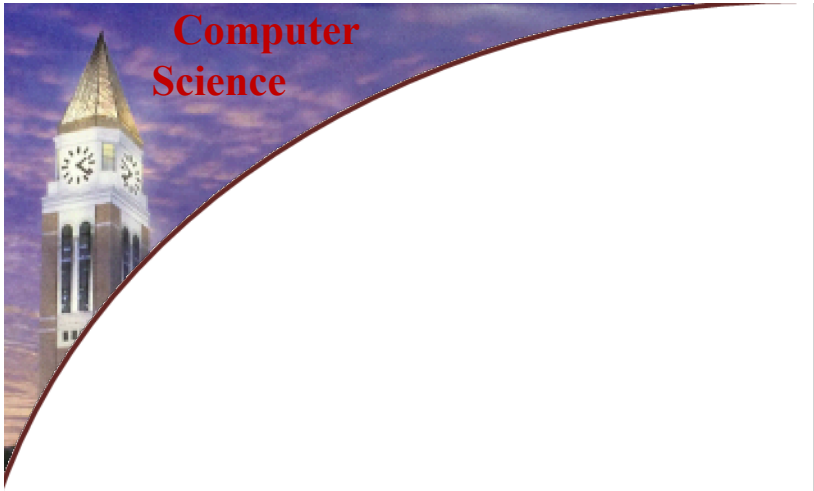
```
(run "{
        val x = 42
        x
        val y = 33
      }"))
```

An <var-expr> cannot follow an <expr>



What is the Abstract Syntax Tree For the following?

```
"{
  val x = 42
  {
   val x = 23
  }
  x
 }"
```



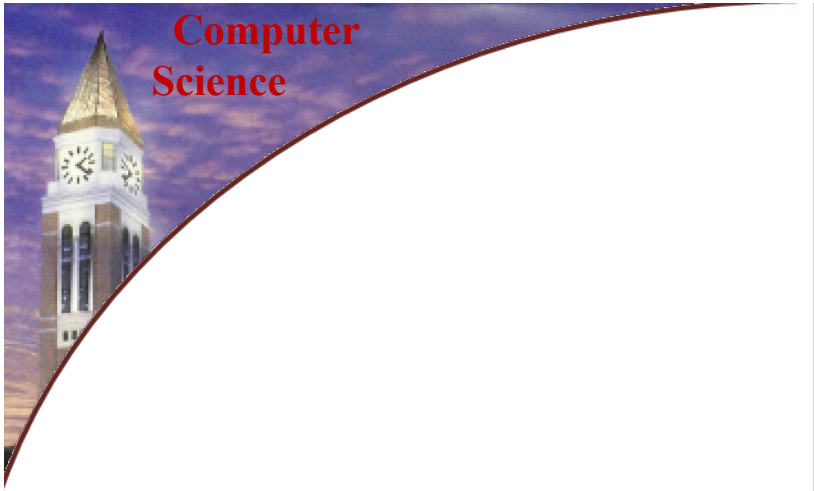
HW06

(Problem 4)

Further hints (code skeleton)

```
(define (run program-string)
  (if (string? program-string)

      (raise (string-append "expected a program as string, got: " (~a program-string)))
      )
  )
```

; to kick off the interpreter here !

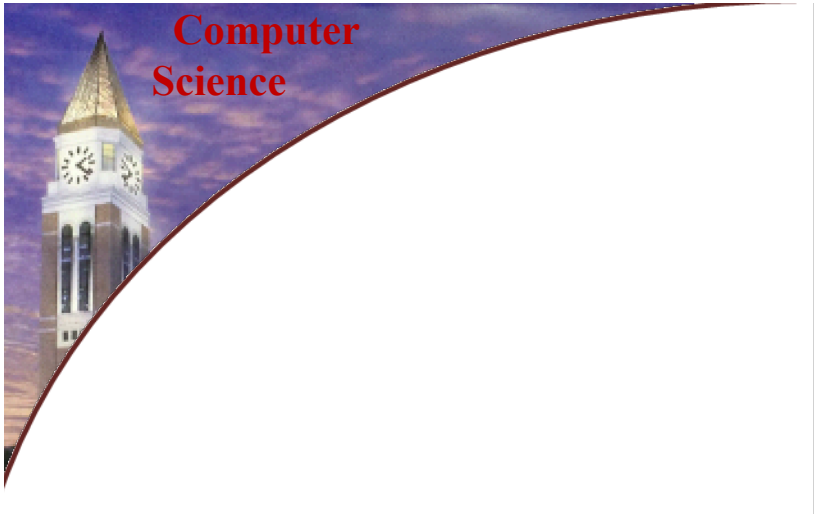


HW06

(Problem 4)

Further hints (code skeleton)

```
(define (run program-string)
  (if (string? program-string)
      (value-of (parser program-string) (empty-env))   ; to kick off the interpreter here !
      (raise (string-append "expected a program as string, got: " (~a program-string)))
      )
  )
```

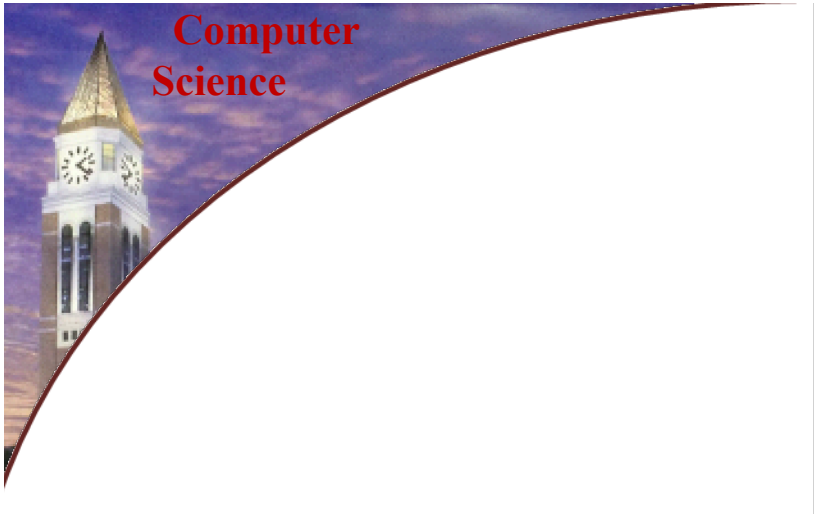


HW06

(Problem 4)

Further hints (code skeleton)

```
(define (run program-string)
  (if (string? program-string)
      (value-of (parser program-string) (empty-env))   ; to kick off the interpreter here !
      (raise (string-append "expected a program as string, got: " (~a program-string)))
      )
  )
```

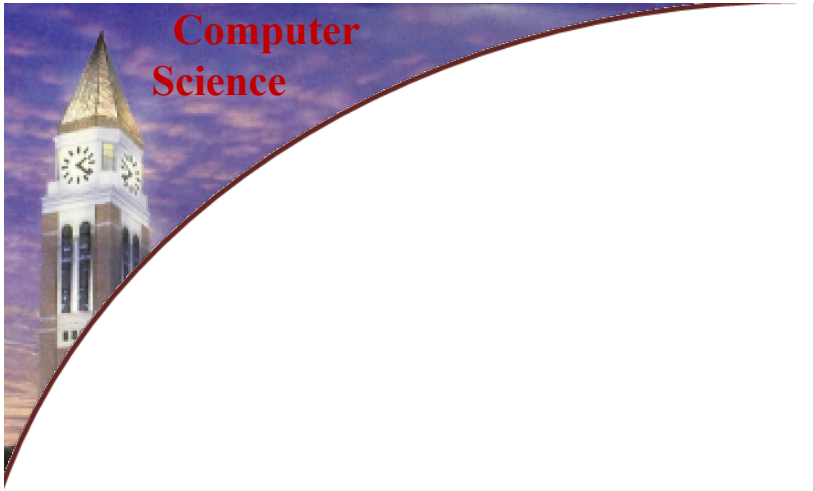


HW06

(Problem 4)

Further hints (code skeleton)

```
(define (run program-string)
  (if (string? program-string)
      (value-of (parser program-string) (empty-env))   ; to kick off the interpreter here !
      (raise (string-append "expected a program as string, got: " (~a program-string)))
      )
  )



(define (value-of ast env)




  )
```

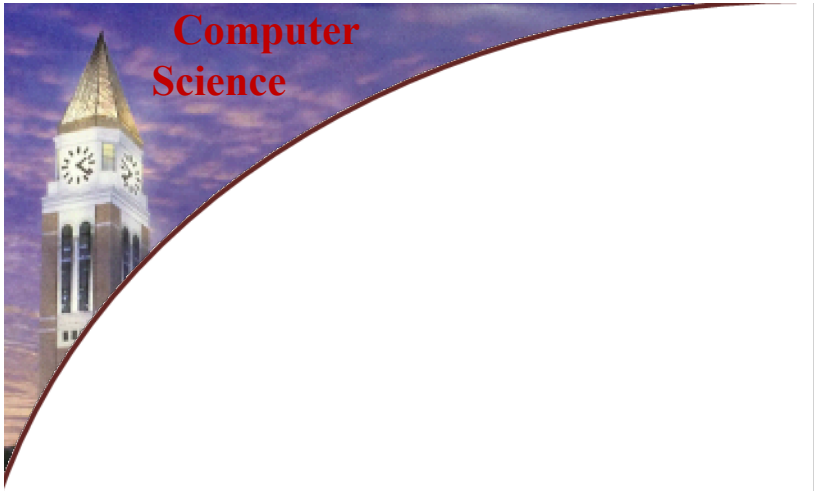


HW06

(Problem 4)

Further hints (code skeleton)

```
(define (run program-string)
  (if (string? program-string)
      (value-of (parser program-string) (empty-env))   ; to kick off the interpreter here !
      (raise (string-append "expected a program as string, got: " (~a program-string)))
      )
  )



(define (value-of ast env)
  (cond
    [(program? ast) (value-of-program ast env)]
    [(expr? ast) (value-of-expr ast env)]
    [(var-expr? ast) (value-of-var ast env)]


  )
```



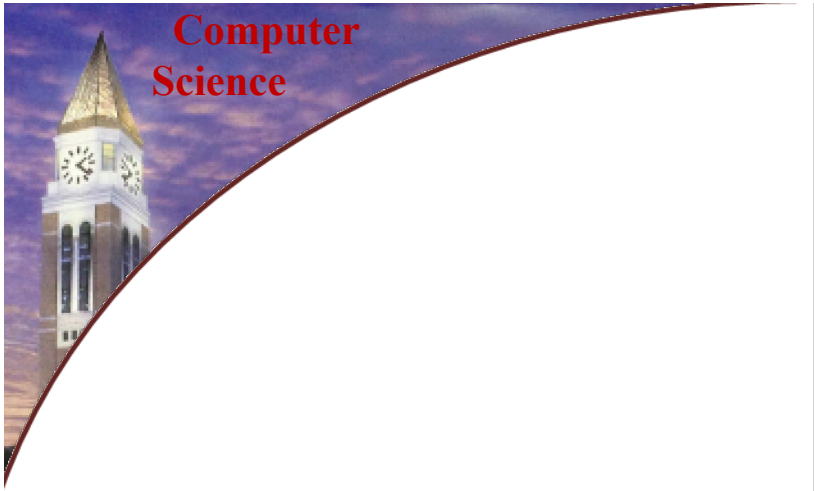
HW06

(Problem 4)

Further hints (code skeleton)

```
(define (run program-string)
  (if (string? program-string)
      (value-of (parser program-string) (empty-env))   ; to kick off the interpreter here !
      (raise (string-append "expected a program as string, got: " (~a program-string)))
      )
  )
```

```
(define (value-of ast env)
  (cond
    [(program? ast) (value-of-program ast env)]
    [(expr? ast) (value-of-expr ast env)]
    [(var-expr? ast) (value-of-var ast env)]
    [else (raise (~a "Unimplemented ast node: " ~a ast))]
    )
  )
```



HW06

(Problem 4)

Further hints (code skeleton)

```
(define (run program-string)
  (if (string? program-string)
      (value-of (parser program-string) (empty-env))
      (raise (string-append "expected a program as string, got: " (~a program-string)))
      )
  )

(define (value-of-program prog env)
  (cases program prog
    (a-program
     (expr rest-of-expressions)
     ;given a non-predicate function, andmap will apply the function
     ;to every element in the list and then return the value of
     ;applying the function on the last element.
     (andmap (lambda (ex) (value-of ex env))
             (flat-list expr rest-of-expressions))
     )
    )
  )
```

; to kick off the interpreter here !

HW06

(Problem 3)

```
<program> ::=
              <expr> <expr>* a-program
<expr> ::=
            number                "num-expr"
          | up(<expr>)            "up-expr"
          | down(<expr>)          "down-expr"
          | left(<expr>)          "left-expr"
          | right(<expr>)         "right-expr"

          | (<expr> <expr>)       "point-expr"
          | + <expr> <expr>       "add-expr"
          | origin? (<expr>)      "origin-expr"
          | if (<expr>)
             then <expr>
             else <expr>           "if-expr"

          | move (<expr> <expr> <expr>*)   "move-expr"
```

HW06

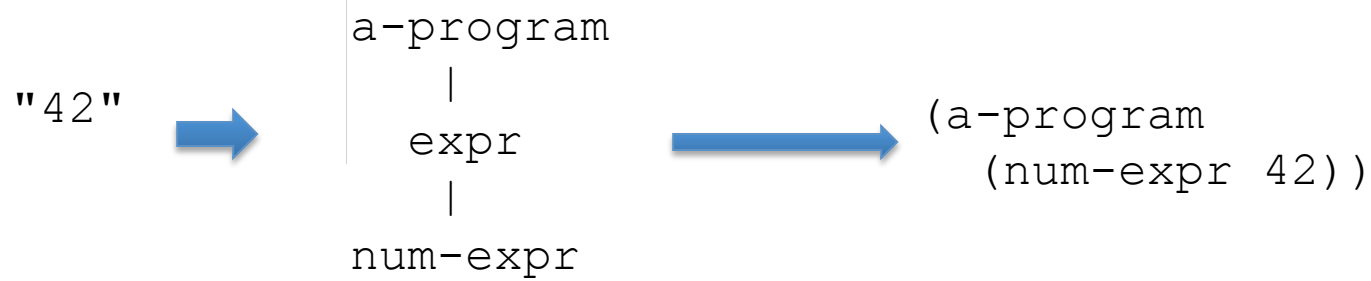
(Problem 3)

```
<program> ::=
              <expr> <expr>* a-program
<expr> ::=
            number              "num-expr"
          | up(<expr>)          "up-expr"
          | down(<expr>)        "down-expr"
          | left(<expr>)        "left-expr"
          | right(<expr>)       "right-expr"
```

```
(define-datatype program program?
        (a-program (first-exp expr?) (rest (list-of expr?)))

(define-datatype expr expr?
        (num-expr (n number?))
        (up-expr  (s expr?))
        (down-expr (s expr?))
        (left-expr (s expr?))
        (right-expr (s expr?)))
```

a-program

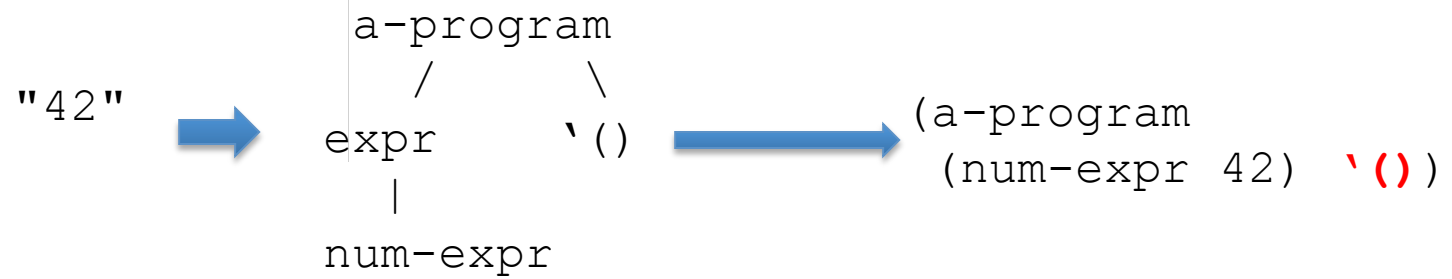
- What is the AST for "42" ?




```
<program> ::=
                <expr> <expr>*   a-program
<expr> ::=
              number                "num-expr"
            | up(<expr>)            "up-expr"
            | down(<expr>)          "down-expr"
            | left(<expr>)          "left-expr"
            | right(<expr>)         "right-expr"
```

a-program

- What is the AST for "42" ?




```
<program> ::=
                <expr> <expr>*   a-program
<expr> ::=
              number                "num-expr"
            | up(<expr>)            "up-expr"
            | down(<expr>)          "down-expr"
            | left(<expr>)          "left-expr"
            | right(<expr>)         "right-expr"
```

a-program

- What is the AST for "42"?


```
<program> ::=
                <expr> <expr>*   a-program
<expr> ::=
              number                "num-expr"
            | up(<expr>)            "up-expr"
            | down(<expr>)          "down-expr"
            | left(<expr>)          "left-expr"
            | right(<expr>)         "right-expr"
```

a-program

- What is the AST for "42"?


```
<program> ::=
                <expr> <expr>*   a-program
<expr> ::=
              number                "num-expr"
            | up(<expr>)            "up-expr"
            | down(<expr>)          "down-expr"
            | left(<expr>)          "left-expr"
            | right(<expr>)         "right-expr"
```

a-program

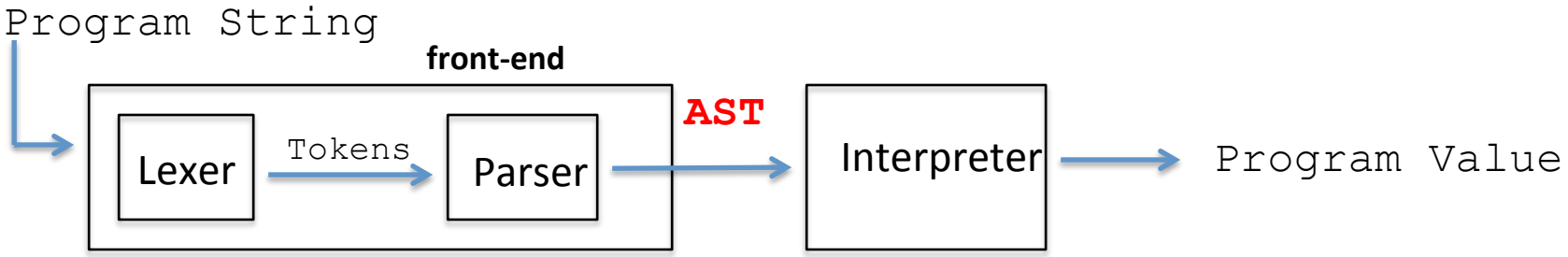
- What is the AST for "42"?

What should (run "42") return?

a-program

- What is the AST for "42"?

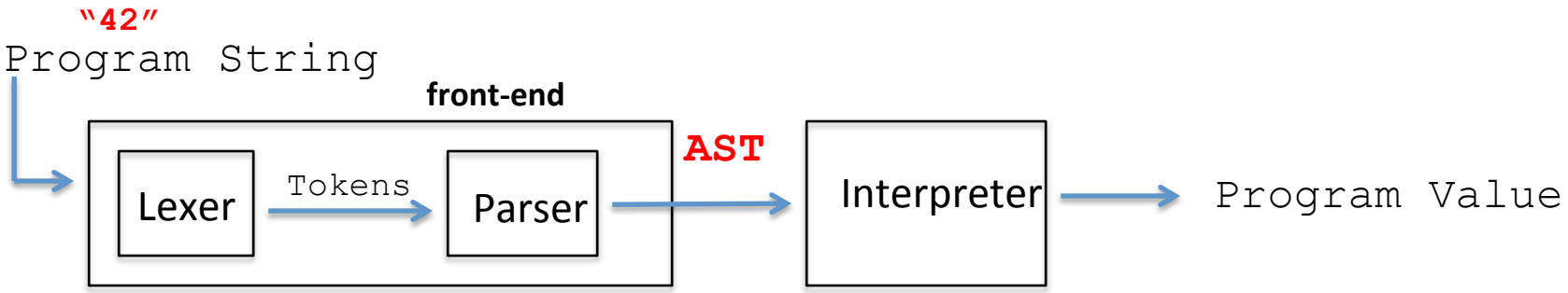
What should (run "42") return?



a-program

- What is the AST for "42"?

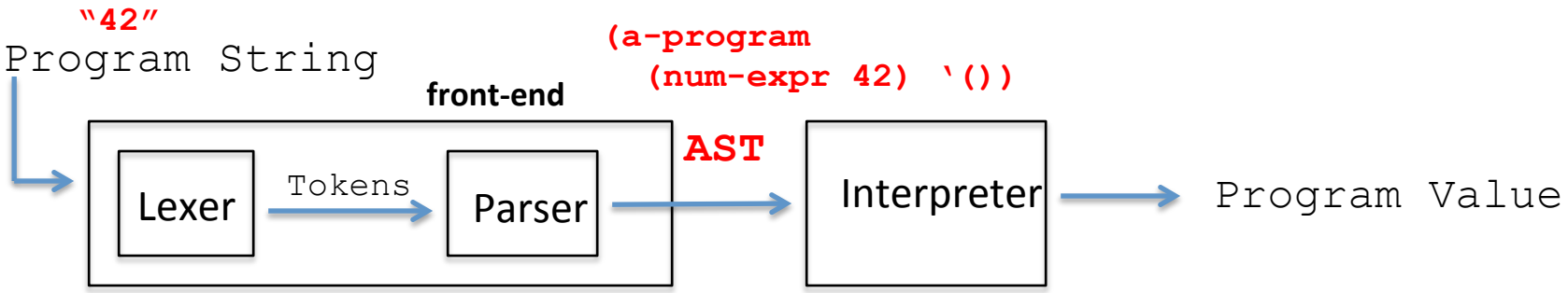
What should (run "42") return?



a-program

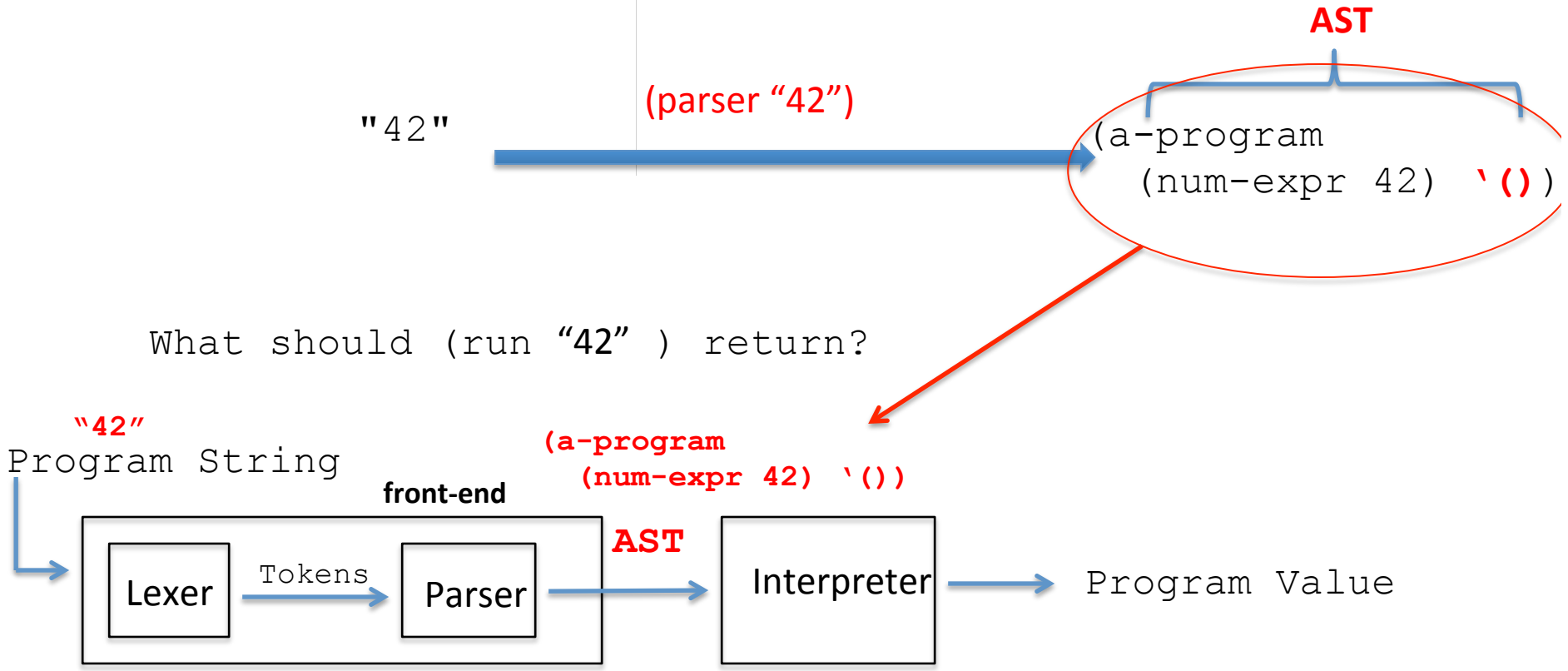
- What is the AST for "42"?

What should (run "42") return?



a-program

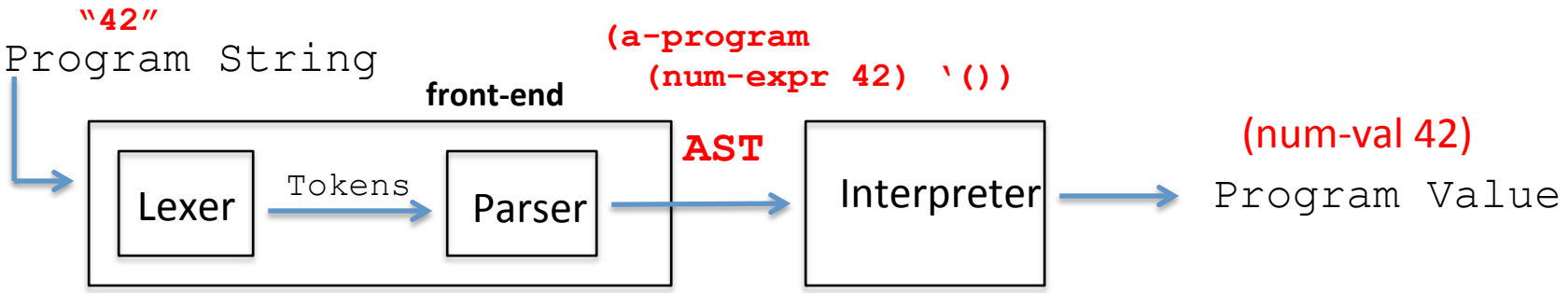
- What is the AST for "42"?



a-program

- What is the AST for "42"?

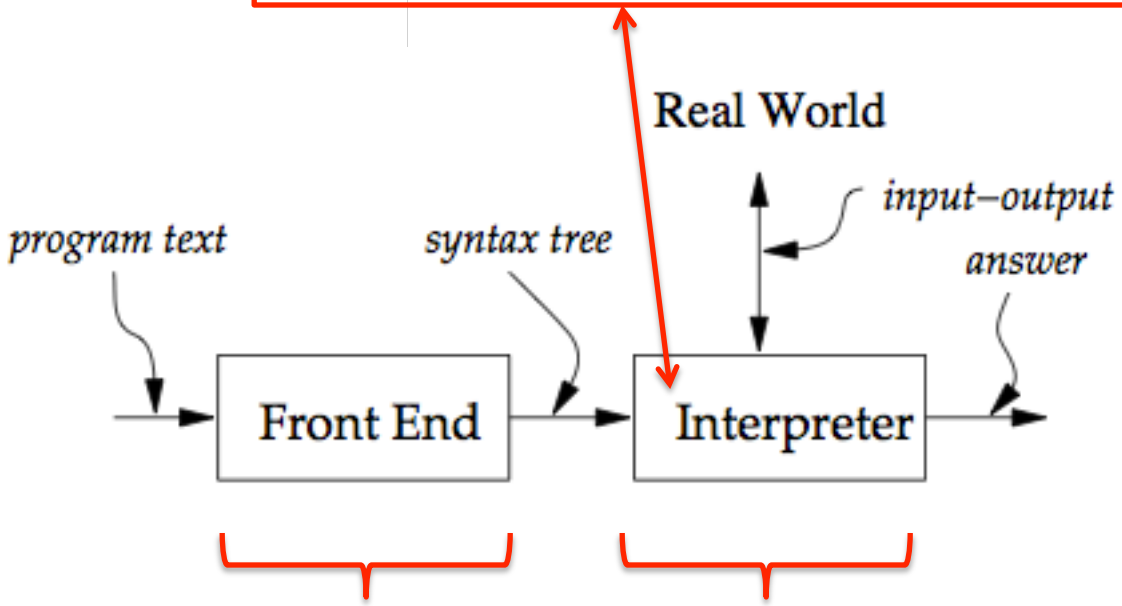
What should (run "42") return?



The Basic Form Of The Interpreter

problem 3

```
(value-of-program ast) => expressled-val
```



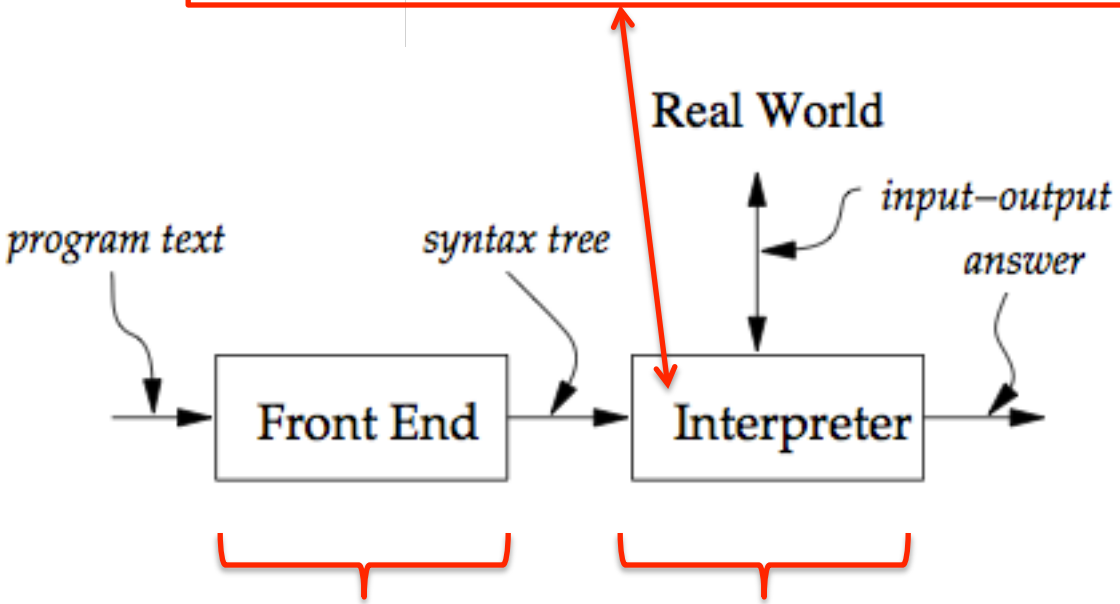
automated *scanning* & *parsing* based on lexical and grammar spec

majority of your work here!

The Basic Form Of The Interpreter

problem 4

```
(value-of-program ast env) => expressled-val
```



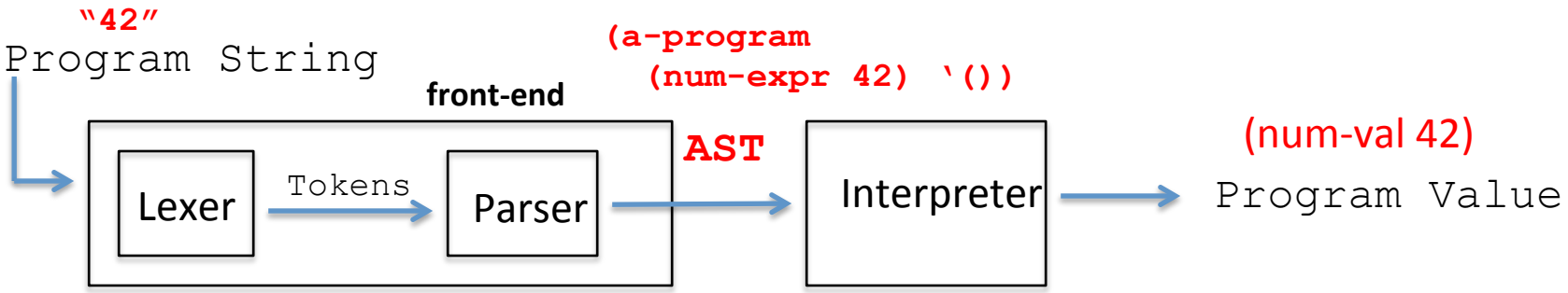
automated *scanning & parsing* based on lexical and grammar spec

majority of your work here!

a-program

- What is the AST for "42"?

What should (run "42") return?

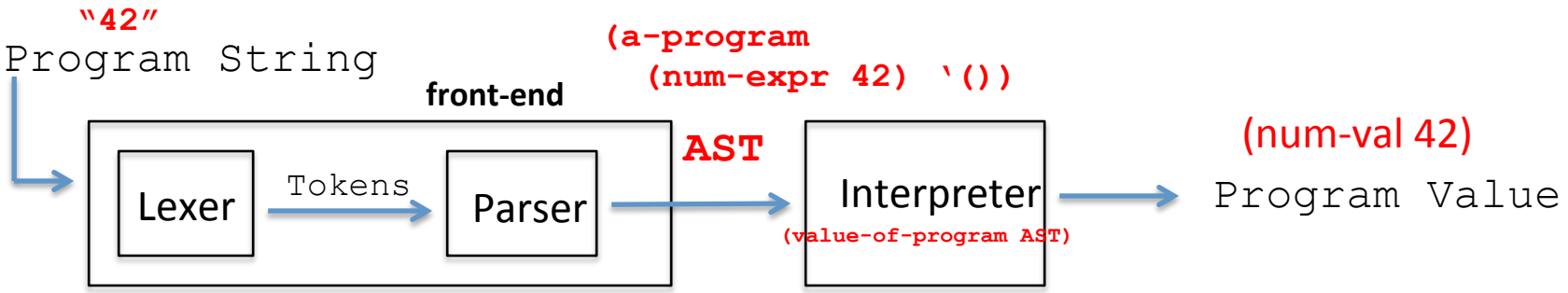


a-program

HW06
Problem 3

- What is the AST for "42"?

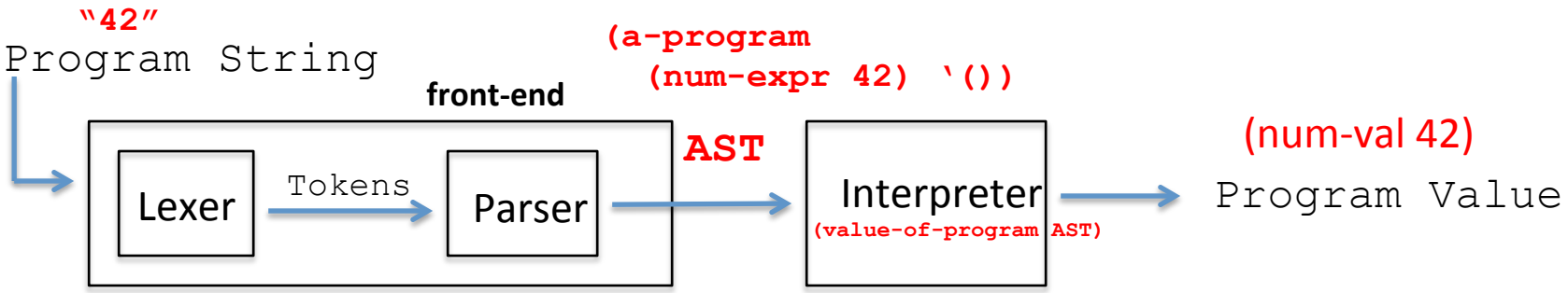
What should (run "42") return?



Intermediate steps

HW06 Problem 3

- (value-of-program ast)

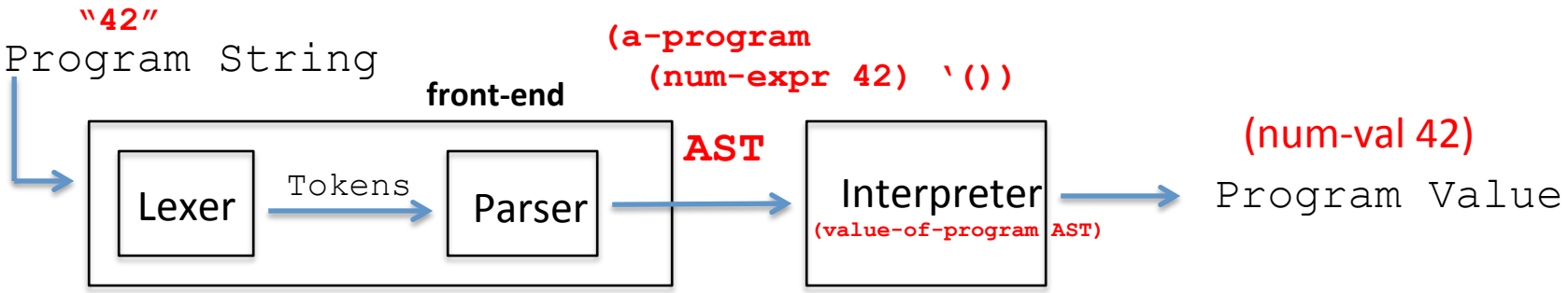


Intermediate steps

HW06 Problem 3

- (value-of-program ast)

replace `ast` with `(a-program (num-expr 42) '() )`



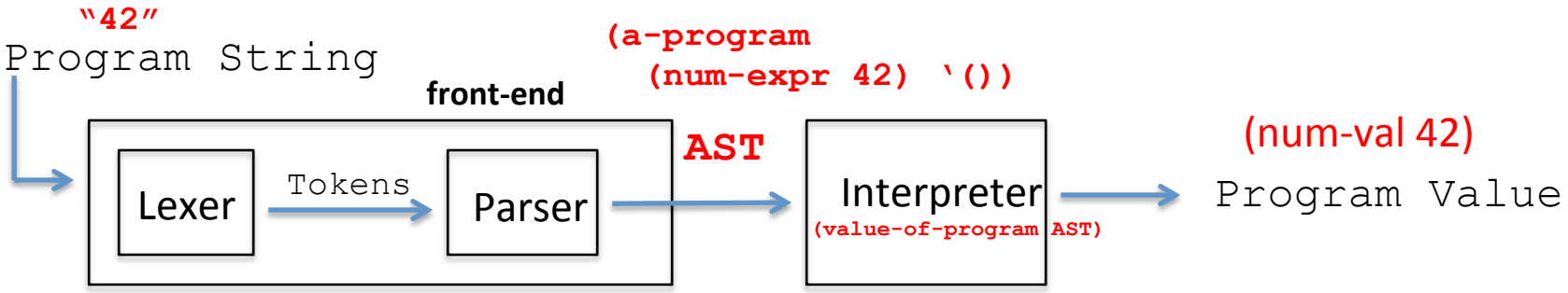
Intermediate steps

HW06 Problem 3

- (value-of-program ast)

automatically by (parser "42")

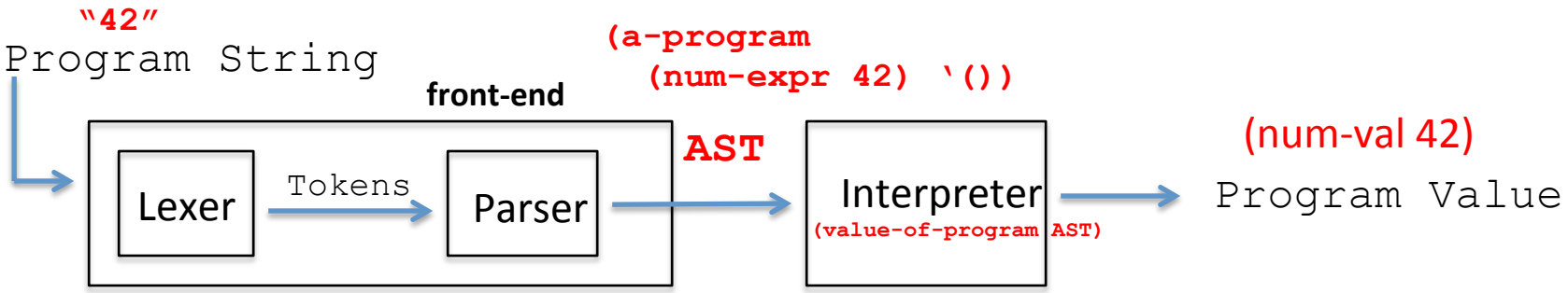
replace `ast` with `(a-program (num-expr 42) '() )`



Intermediate steps

HW06 Problem 3

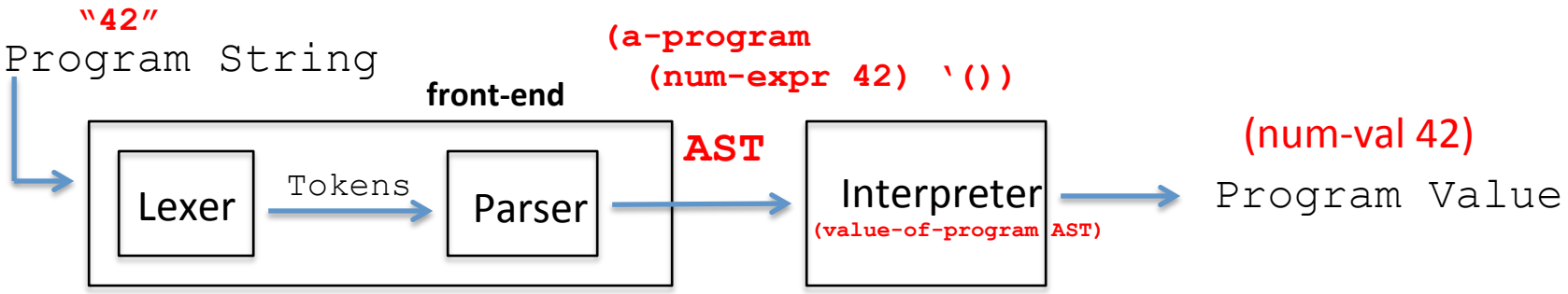
```
(value-of-program (a-program (num-expr 42) '()) )
```



Intermediate steps

HW06 Problem 3

```
(value-of-program (a-program (num-expr 42) '()) )
```

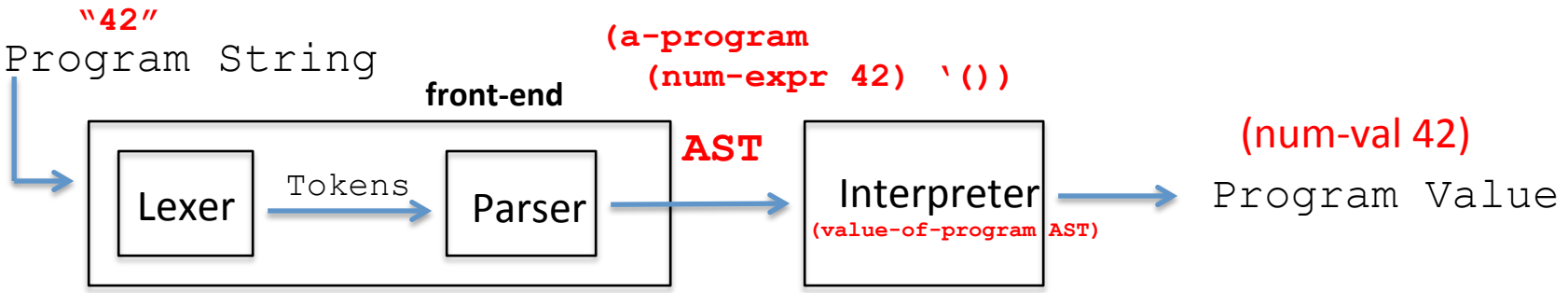


Intermediate steps

HW06 Problem 3

```
(value-of-program (a-program (num-expr 42) '()) )
```

```
(value-of-expr (num-expr 42))
```



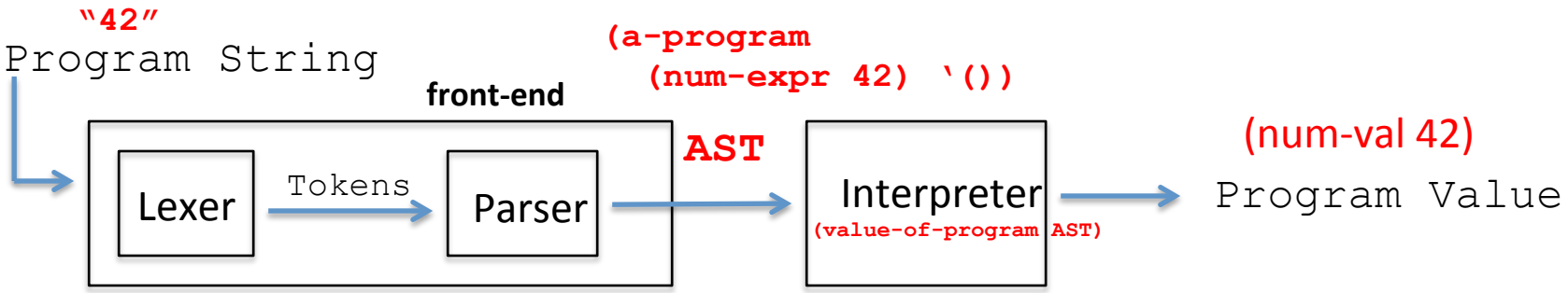
Intermediate steps

HW06 Problem 3

```
(value-of-program (a-program (num-expr 42) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (num-expr 42) '()))

```
(value-of-expr (num-expr 42))
```



Intermediate steps

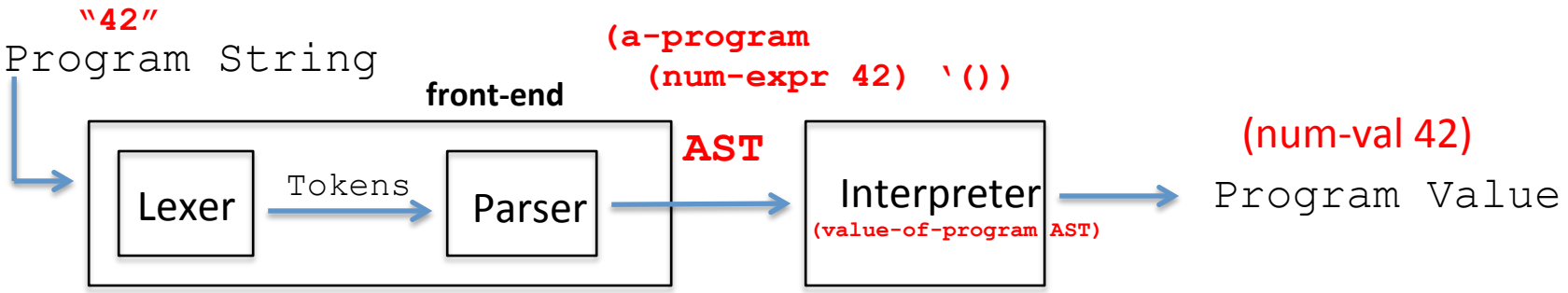
HW06 Problem 3

```
(value-of-program (a-program (num-expr 42) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (num-expr 42) '()))

```
(value-of-expr (num-expr 42))
```

How to generate (num-val 42)
from (num-expr 42) ?



Intermediate steps

HW06 Problem 3

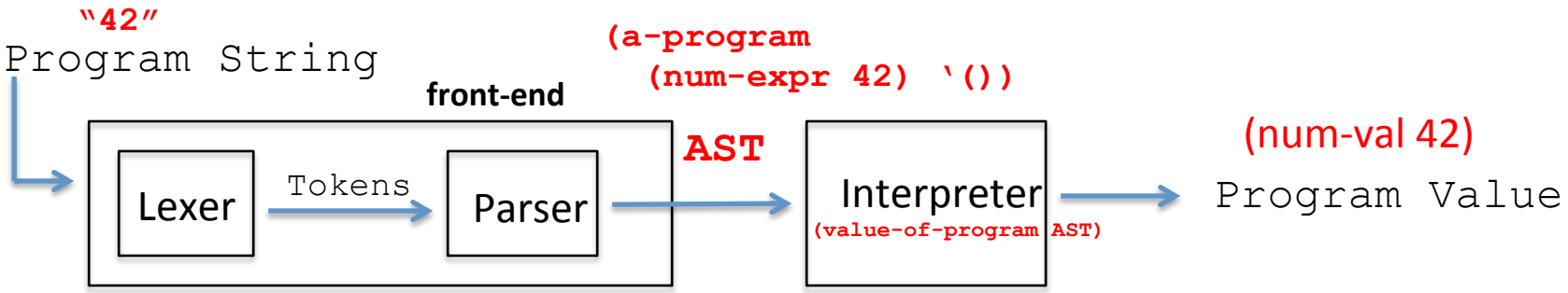
```
(value-of-program (a-program (num-expr 42) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (num-expr 42) '()))

```
(value-of-expr (num-expr 42))
```

How to generate (num-val 42)
from (num-expr 42) ?

Use **cases** !



Intermediate steps

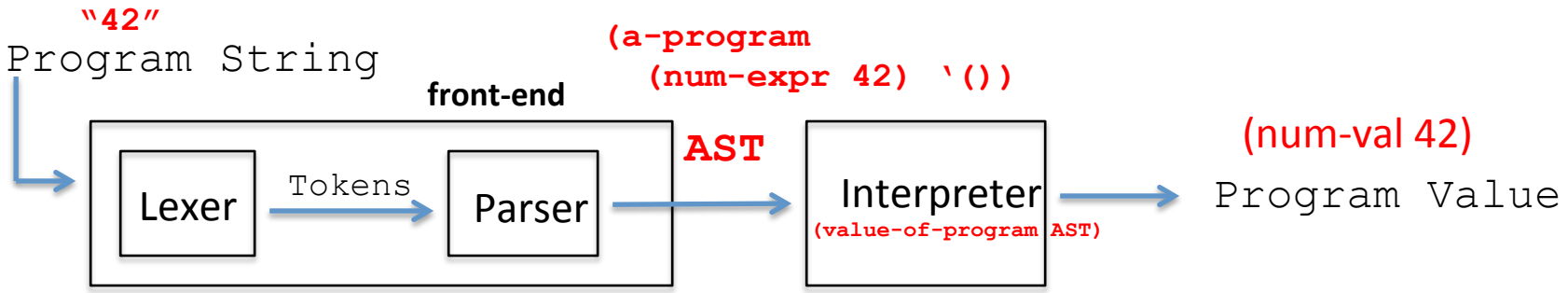
HW06 Problem 3

```
(value-of-program (a-program (num-expr 42) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (num-expr 42) '()))

```
(value-of-expr (num-expr 42))
```

```
(define (value-of-expr e)
 (cases expr e
   (num-expr (n) ...) )
```



Intermediate steps

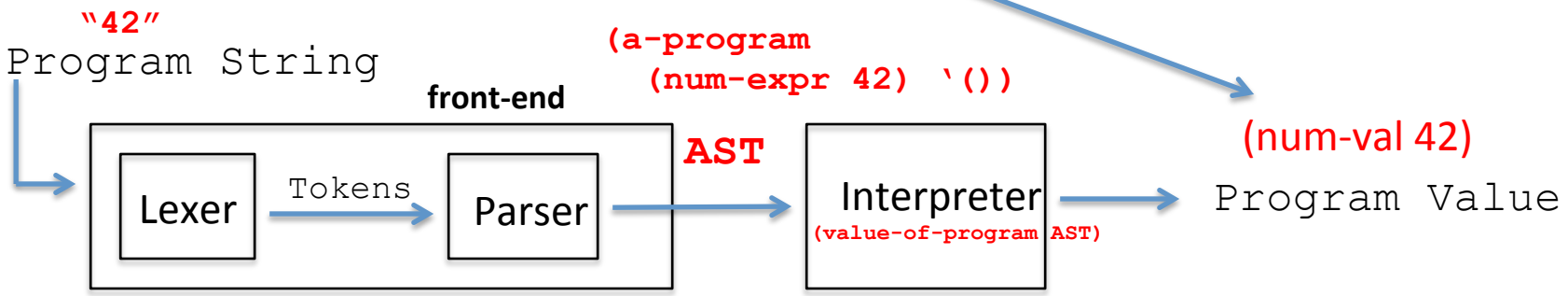
HW06 Problem 3

```
(value-of-program (a-program (num-expr 42) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (num-expr 42) '()))

```
(value-of-expr (num-expr 42))
```

```
(define (value-of-expr e)
 (cases expr e
   (num-expr (n) ...) )
```



Intermediate steps

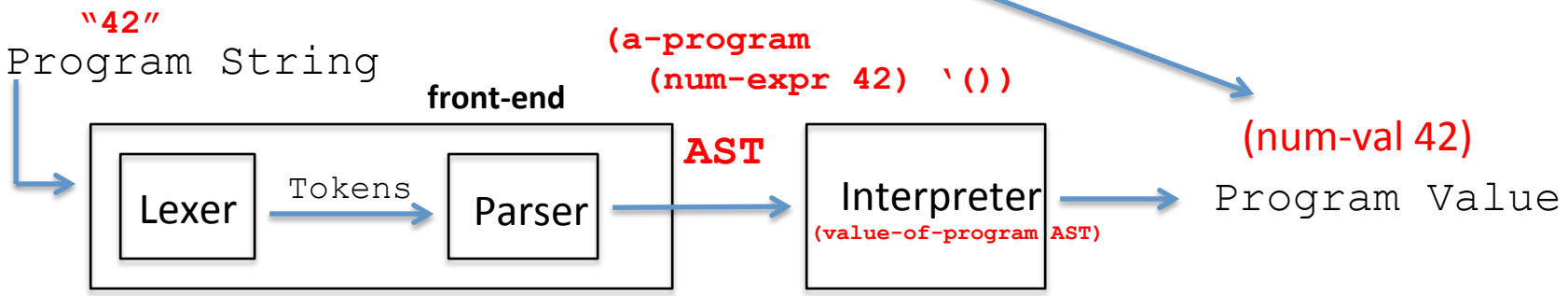
HW06 Problem 3

```
(value-of-program (a-program (num-expr 42) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (num-expr 42) '()))

```
(value-of-expr (num-expr 42))
```

```
(define (value-of-expr e)
 (cases expr e
   (num-expr (n) (num-val n) ) ) )
```



a-program

HW06
Problem 3

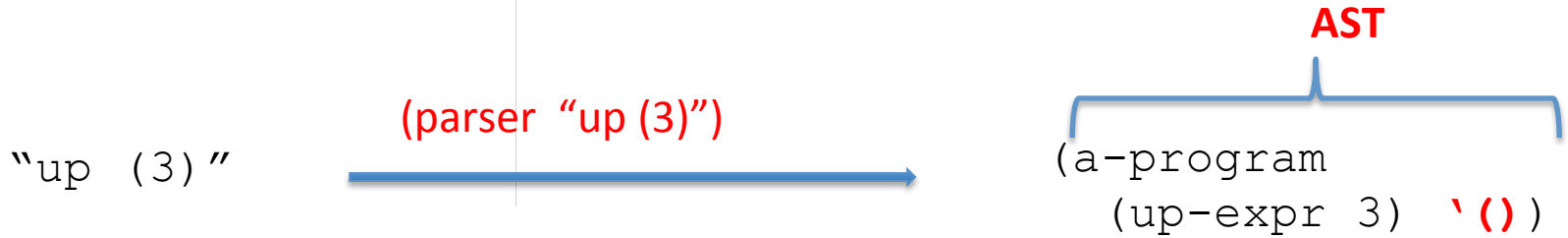
- How about this?

```
(run "up (3)")
```

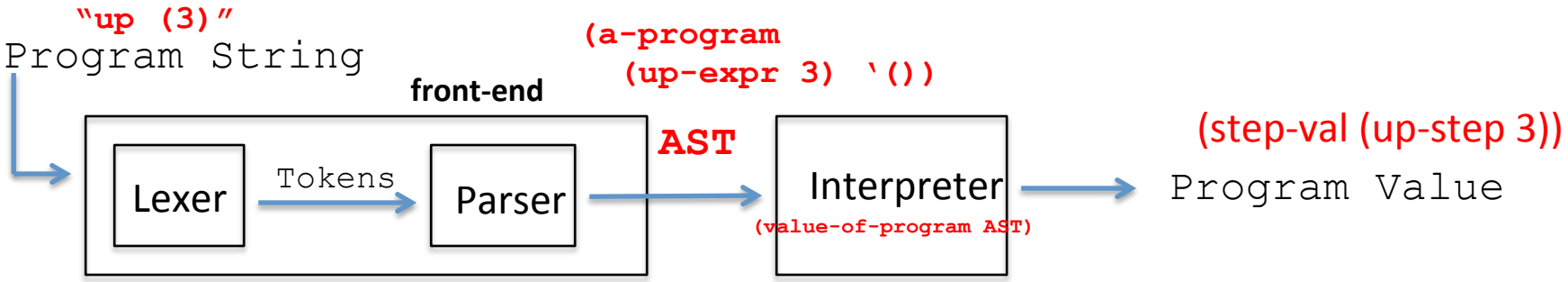
a-program

HW06 Problem 3

- What is the AST for "`up (3)`"?



What should (run "up (3)") return?

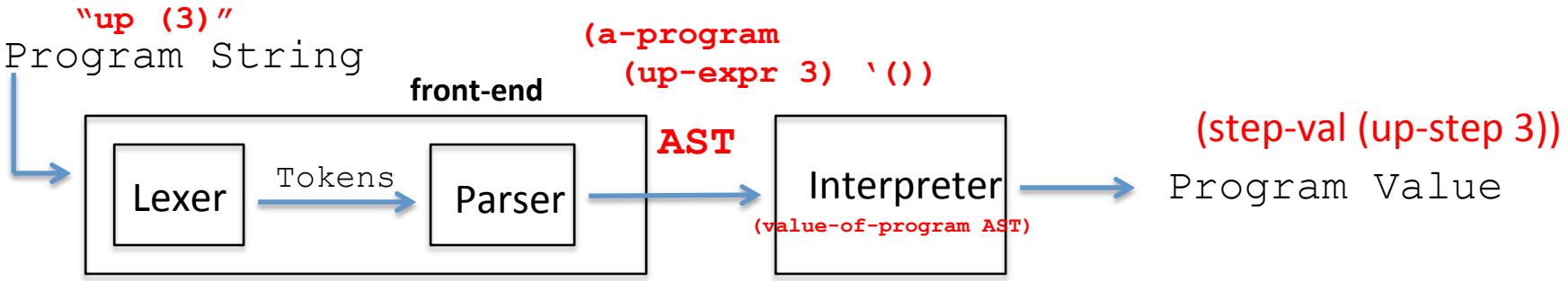


Intermediate steps

HW06 Problem 3

```
(value-of-program (a-program (up-expr 3) '()) )
```

```
(value-of-expr (up-expr 3))
```



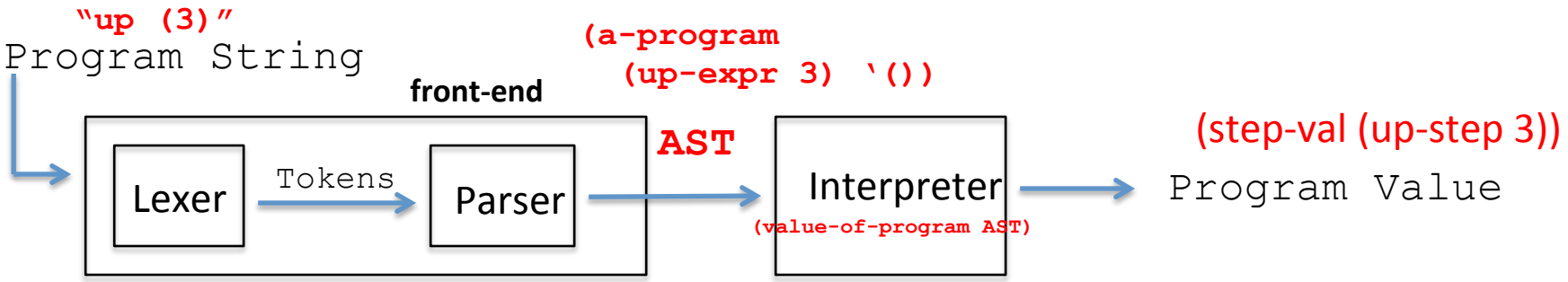
Intermediate steps

HW06 Problem 3

```
(value-of-program (a-program (up-expr 3) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (up-expr 3) '()))

```
(value-of-expr (up-expr 3))
```



Intermediate steps

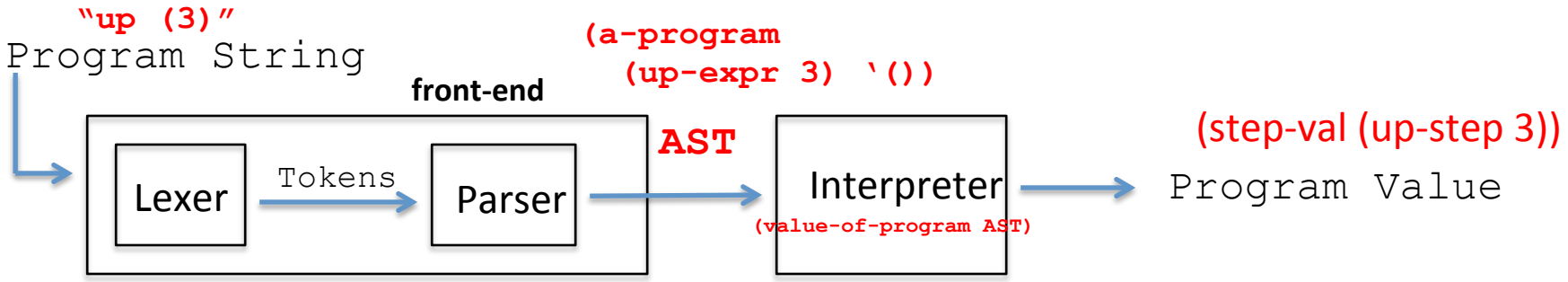
HW06 Problem 3

```
(value-of-program (a-program (up-expr 3) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (up-expr 3) '()))

```
(value-of-expr (up-expr 3))
```

```
(define (value-of-expr e)
 (cases expr e
   (up-expr (n) ... ) ) )
```



Intermediate steps

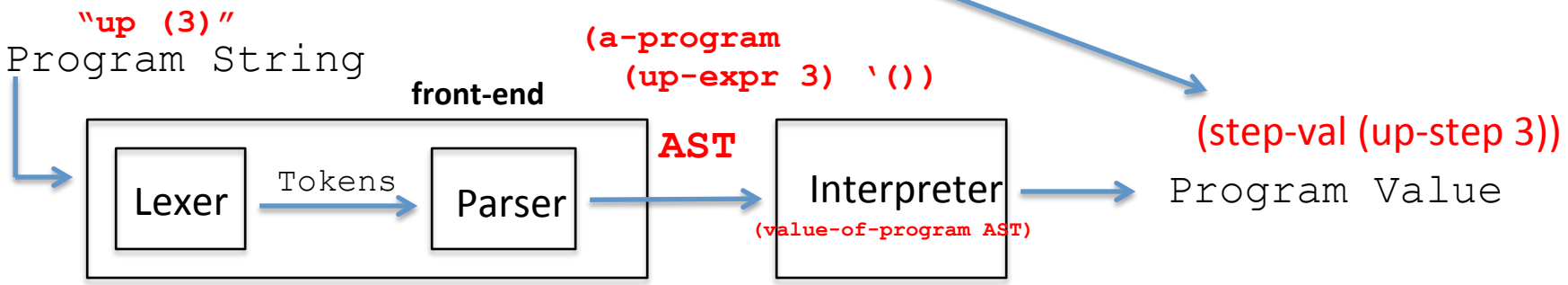
HW06 Problem 3

```
(value-of-program (a-program (up-expr 3) '()) )
```

(andmap (lambda(x) (value-of-expr x))
(flat-list (up-expr 3) '()))

```
(value-of-expr (up-expr 3))
```

```
(define (value-of-expr e)
 (cases expr e
   (up-expr (n) (step-val (up-step n)) ) ) )
```



a-program

- How about this?

```
(run "42
      up(3)
      #comment
      down(4)
      111")
```

a-program

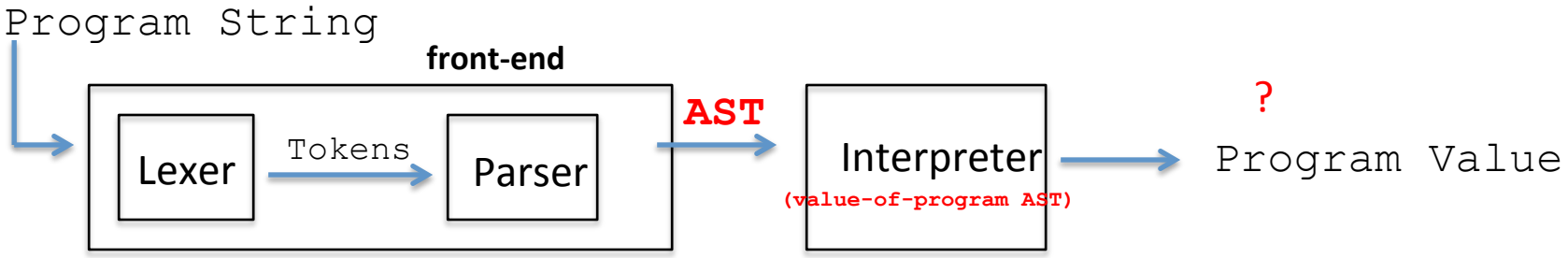
HW06 Problem 3

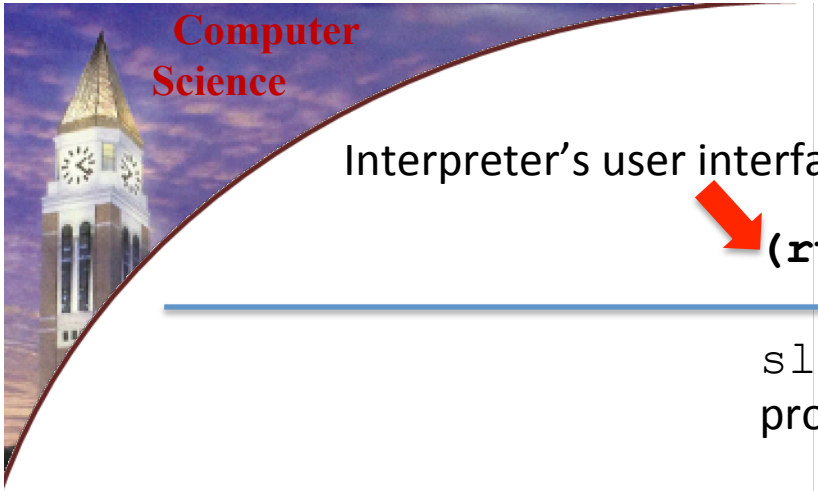
```
"42
 up(3)
 #comment
 down(4)
 111")
```

(parser "42 up(3) # commen down(4) 111")

AST

```
(a-program
  (num-expr 42)
  (list (up-expr (num-expr 3))
        (down-expr (num-expr 4))
        (num-expr 111)))
```





Interpreter's user interface

`(run program-string)`

HW06
Problem 4

`sllgen` generates parser that turns program-string into `ast`

(value-of `ast` **env**)

- (program? `ast`)
- (expr? `ast`)
- (var-expr? `ast`)

Back-end

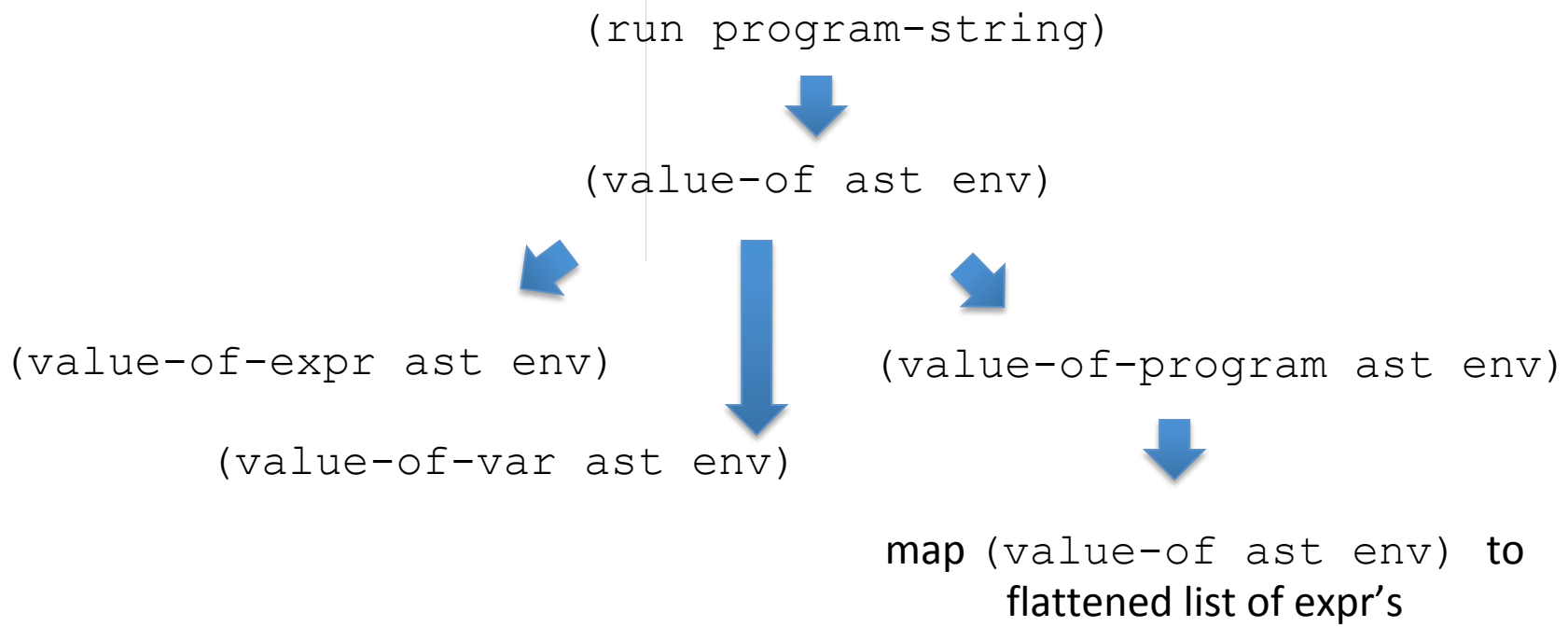
(value-of-program `ast` **env**)

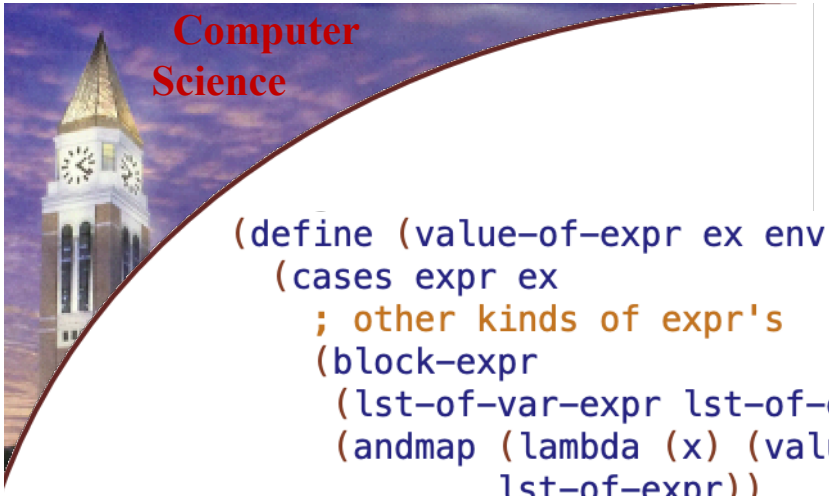
(value-of-expr `ast` **env**)

(value-of-var `ast` **env**)

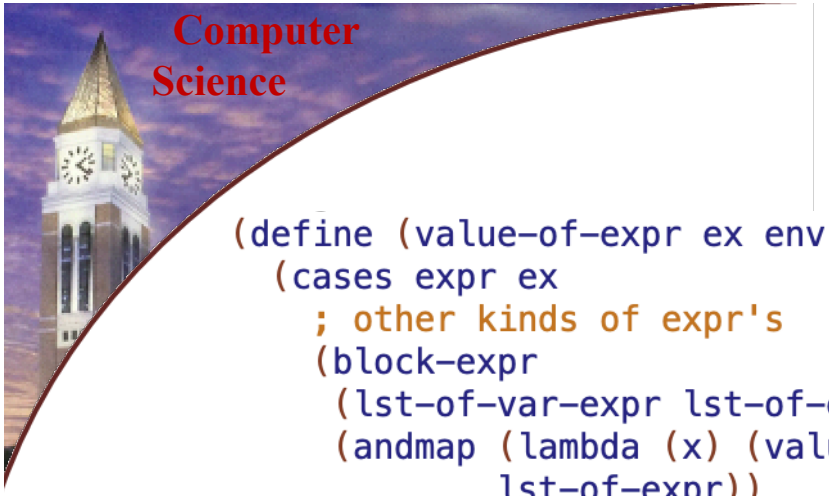
a-program

HW06
Problem 4

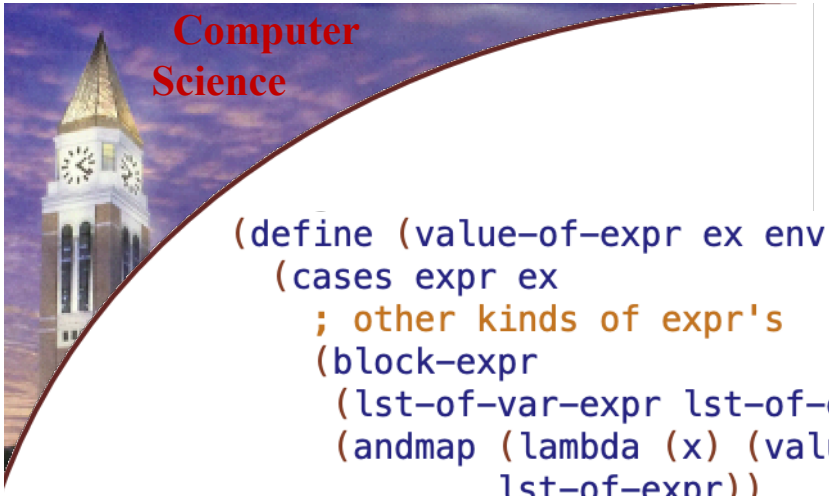





```
(define (value-of-expr ex env)
  (cases expr ex
    ; other kinds of expr's
    (block-expr
     (lst-of-var-expr lst-of-expr)
     (andmap (lambda (x) (value-of x (build-new-env lst-of-var-expr env)))
             lst-of-expr))
    ;...
    ))
```

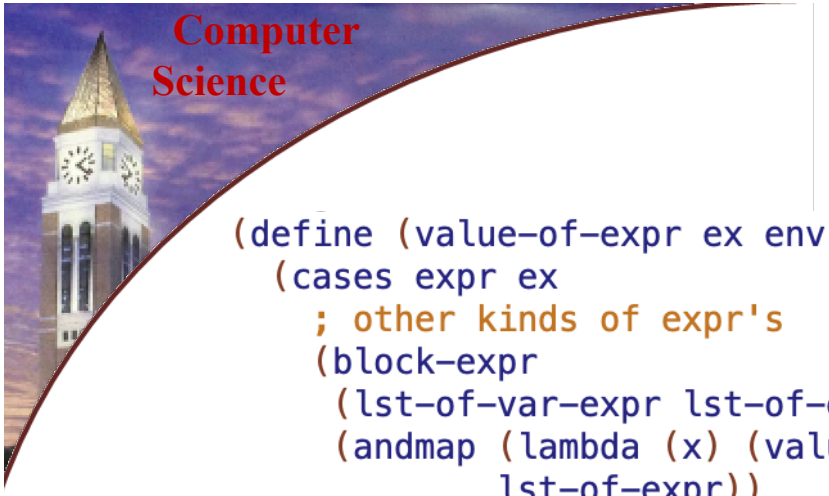



```
(define (value-of-expr ex env)
  (cases expr ex
    ; other kinds of expr's
    (block-expr
     (lst-of-var-expr lst-of-expr)
     (andmap (lambda (x) (value-of x (build-new-env lst-of-var-expr env)))
             lst-of-expr))
    ;...
    ))
```



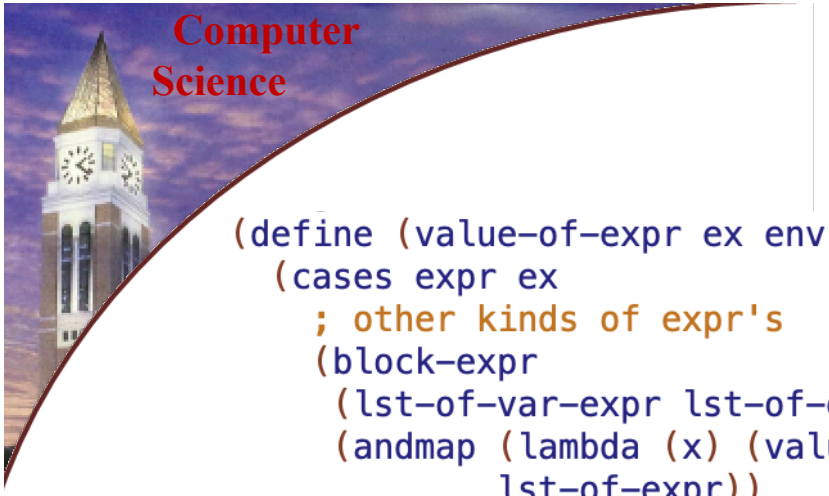

```
(define (value-of-expr ex env)
  (cases expr ex
    ; other kinds of expr's
    (block-expr
     (lst-of-var-expr lst-of-expr)
     (andmap (lambda (x) (value-of x (build-new-env lst-of-var-expr env)))
             lst-of-expr))
    ;...
    ))
```

```
;============================helpers for block-expr==============
(define (build-new-env lst-var-expr old-env)
 (if (null? lst-var-expr)
     old-env
     (build-new-env (cdr lst-var-expr) (one-at-a-time-add (car lst-var-expr) old-env))))
```



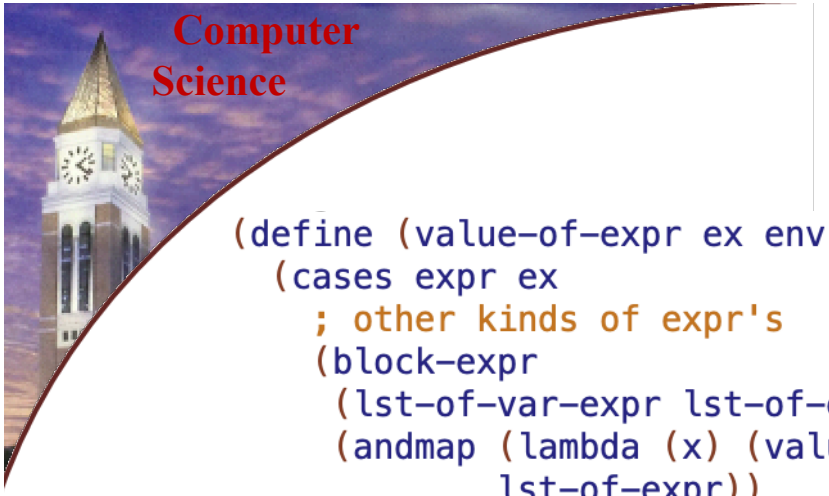

```
(define (value-of-expr ex env)
  (cases expr ex
    ; other kinds of expr's
    (block-expr
     (lst-of-var-expr lst-of-expr)
     (andmap (lambda (x) (value-of x (build-new-env lst-of-var-expr env)))
             lst-of-expr))
    ;...
    ))
```

```
;============================helpers for block-expr==============
(define (build-new-env lst-var-expr old-env)
 (if (null? lst-var-expr)
     old-env
     (build-new-env (cdr lst-var-expr) (one-at-a-time-add (car lst-var-expr) old-env))))
```



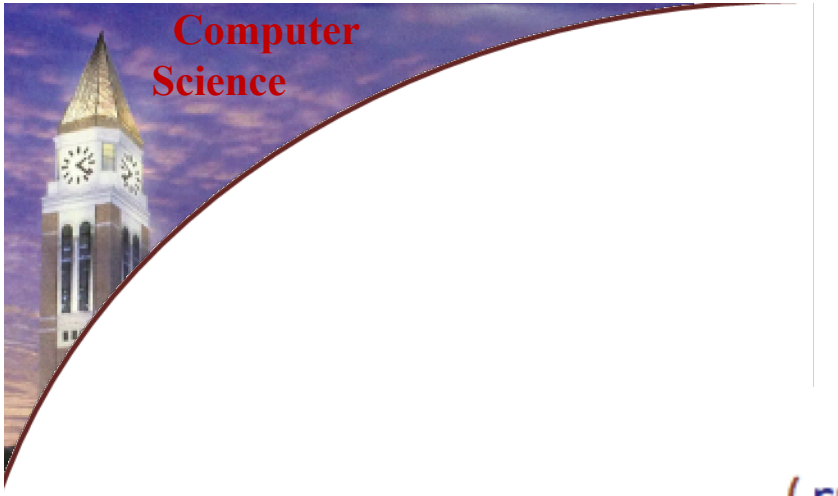

```
(define (value-of-expr ex env)
  (cases expr ex
    ; other kinds of expr's
    (block-expr
     (lst-of-var-expr lst-of-expr)
     (andmap (lambda (x) (value-of x (build-new-env lst-of-var-expr env)))
             lst-of-expr))
    ;...
    ))
```

```
;============================helpers for block-expr==============
(define (build-new-env lst-var-expr old-env)
 (if (null? lst-var-expr)
     old-env
     (build-new-env (cdr lst-var-expr) (one-at-a-time-add (car lst-var-expr) old-env))))
```

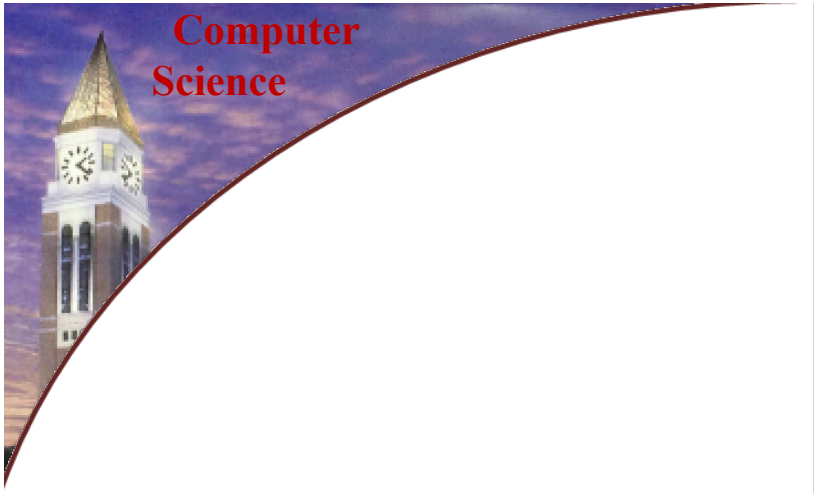



```
(define (value-of-expr ex env)
  (cases expr ex
    ; other kinds of expr's
    (block-expr
     (lst-of-var-expr lst-of-expr)
     (andmap (lambda (x) (value-of x (build-new-env lst-of-var-expr env)))
             lst-of-expr))
    ;...
    ))
```

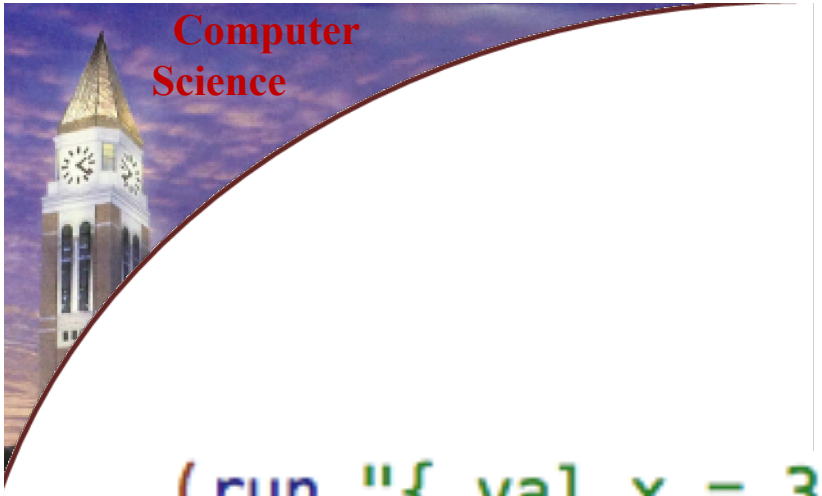
```
;============================helpers for block-expr==============
(define (build-new-env lst-var-expr old-env)
 (if (null? lst-var-expr)
     old-env
     (build-new-env (cdr lst-var-expr) (one-at-a-time-add (car lst-var-expr) old-env))))

(define (one-at-a-time-add var old-env)
  (cases var-expr var
    (val (iden val-of-iden)
         (extend-env-wrapper iden (value-of val-of-iden old-env) old-env NON-FINAL))
    (final-val (iden val-of-iden)
               (extend-env-wrapper iden (value-of val-of-iden old-env) old-env FINAL))))
```



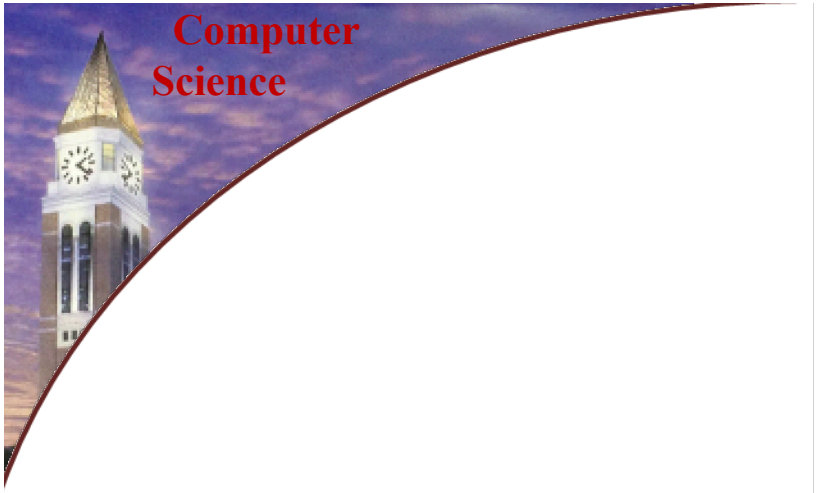

```
(run
 "{
    val x = 42
    {
     val x = 23
    }
    x
  }")
```




```
(run "{
        val x = 42
         {val x = 23
          x
         }
      }" )
```




```
(run "{ val x = 3 }")
```



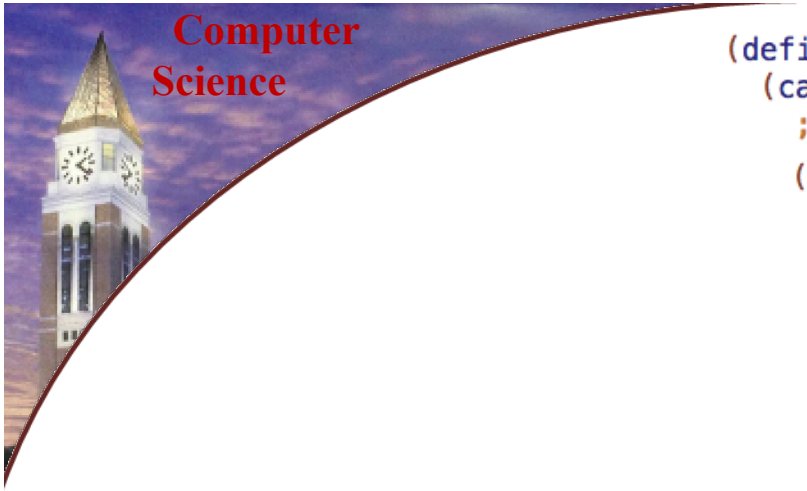

```
"{
  val x = 42
  {
   val x = 23
  }
  x
 }"
```

val-expr

expr (block-expr is a variant of expr)

expr (iden-expr is a variant of expr)

expr (block-expr is a variant of expr)




```
(define (value-of-expr ex env)
  (cases expr ex
    ;other variants of expr
    (iden-expr
     (var-name)
     (apply-env env var-name))
```

```
"{
  val x = 42
  {
   val x = 23
  }
  x
 }"
```

val-expr

expr (block-expr is a variant of expr)

expr (iden-expr is a variant of expr)

expr (block-expr is a variant of expr)


```
(define (value-of-expr ex env)
  (cases expr ex
    ;other variants of expr

    (iden-expr
     (var-name)
     (apply-env env var-name))
```

```
(define (value-of-expr ex env)
  (cases expr ex
    ; other kinds of expr's
    (block-expr
     (lst-of-var-expr lst-of-expr)
     (andmap (lambda (x) (value-of x (build-new-env lst-of-var-expr env)))
             lst-of-expr))
    ;...
    ))
```

```
"{
  val x = 42
  {
   val x = 23
  }
  x
 }"
```

val-expr

expr (block-expr is a variant of expr)

expr (iden-expr is a variant of expr)

expr (block-expr is a variant of expr)

The LET language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`
`zero?-exp (exp1)`

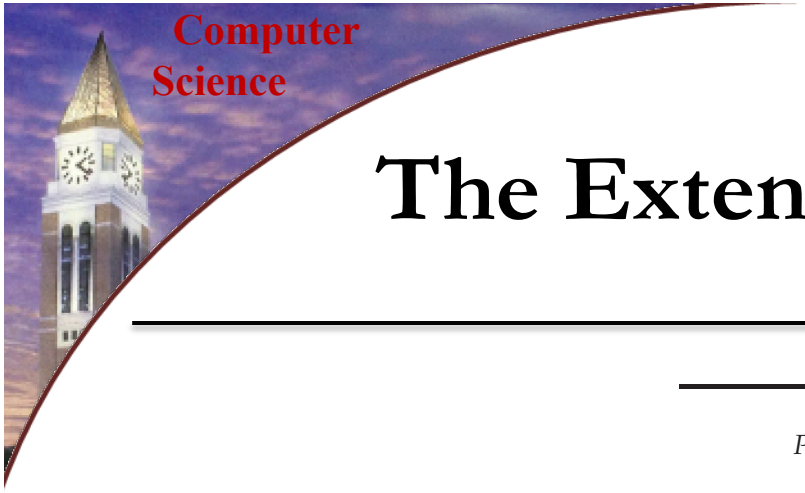
Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Figure 3.2 Syntax for the LET language

How to extend **LET** with **procedure** handling capabilities?



The Extended LET language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

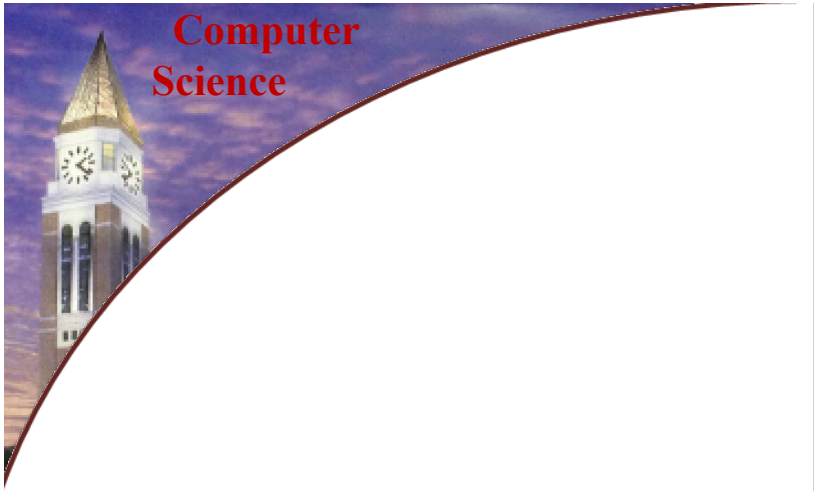
Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Figure 3.2 Syntax for the LET language

How to extend **LET** with **procedure** handling capabilities?

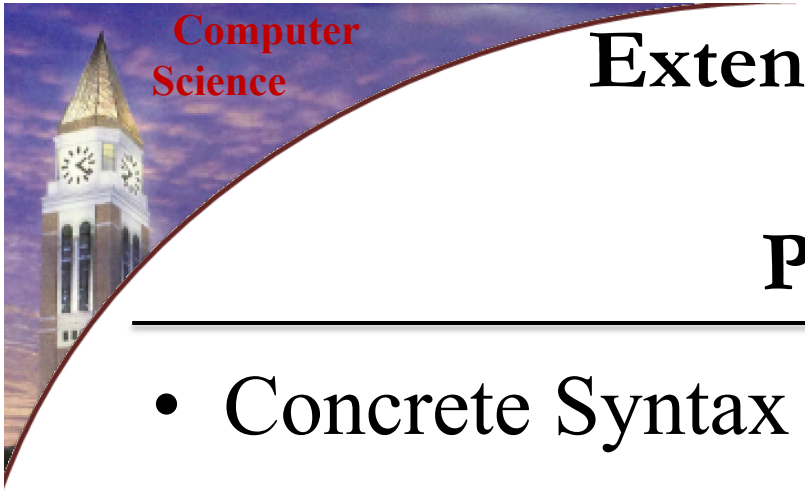
Expression ::= `proc (`*Identifier*`)` *Expression*
`proc-exp (var body)`

Expression ::= `(`*Expression* *Expression*`)`
`call-exp (rator rand)`



Suggested reading:

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)



Extend the language with Procedures

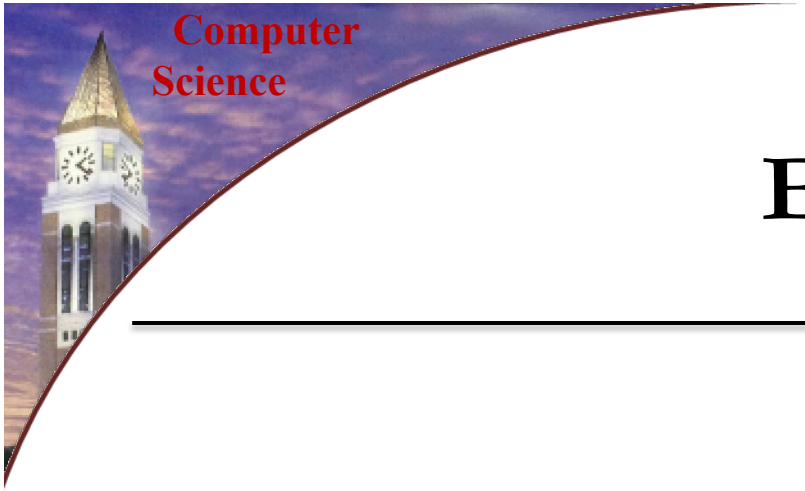
- Concrete Syntax rules -

Procedure Definition

Expression ::= `proc` (Identifier) Expression

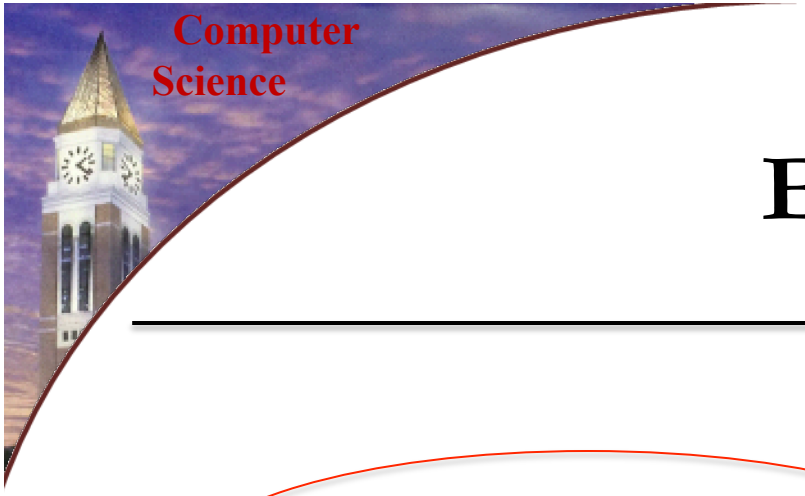
| (Expression Expression)

Procedure Call



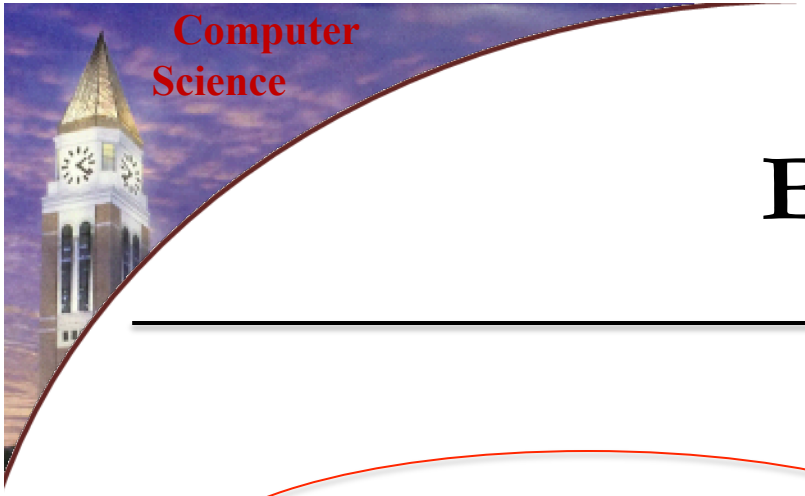
Examples

```
let f = proc (x) -(x,11)
        in (f (f 77))
```



Examples

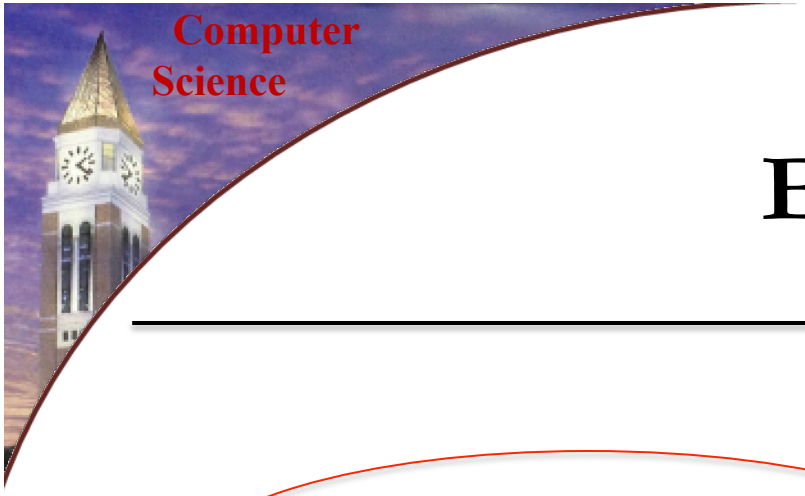
```
let f = proc (x) -(x,11)
        in (f (f 77))
```



Examples

```
let f = proc (x) -(x,11)
        in (f (f 77))
```

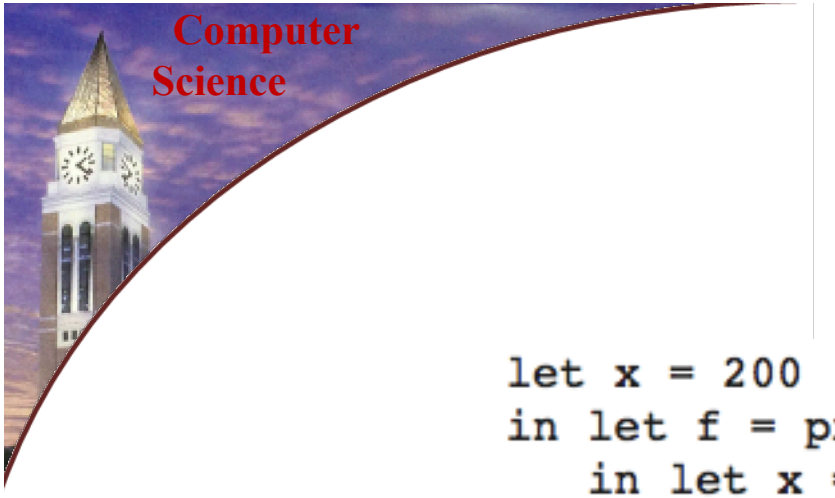
what should be the value of this let expression ?



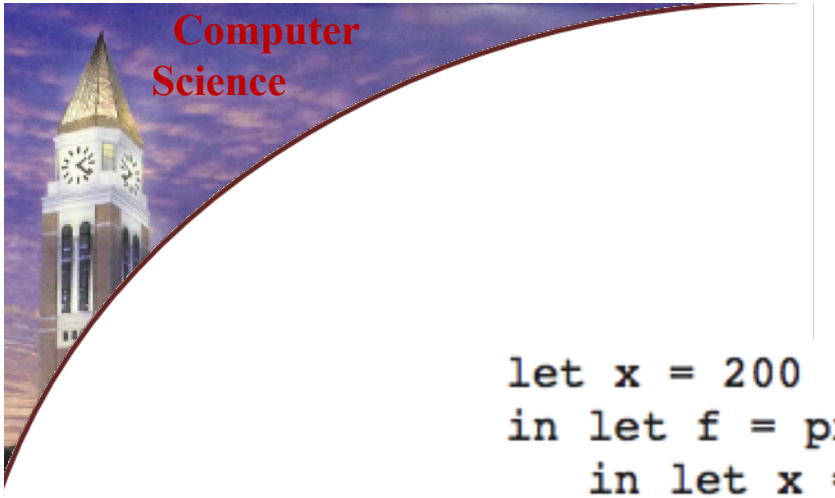
Examples

```
let f = proc (x) -(x,11)
        in (f (f 77))
```

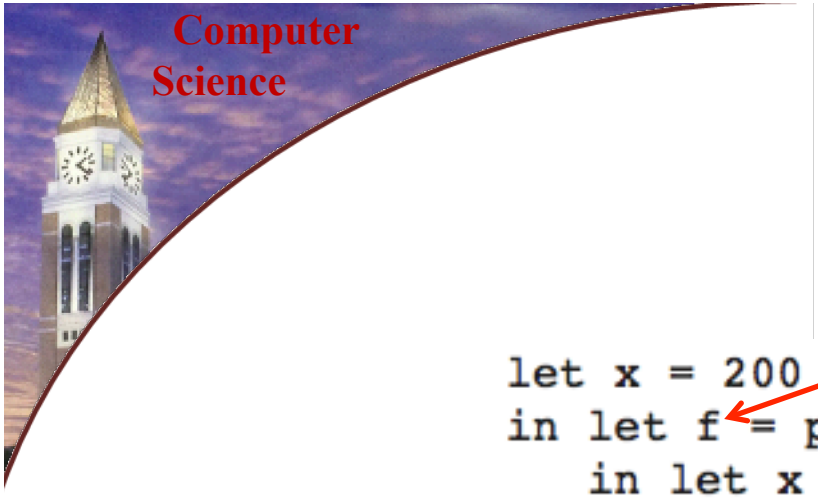
←(num-val 55)



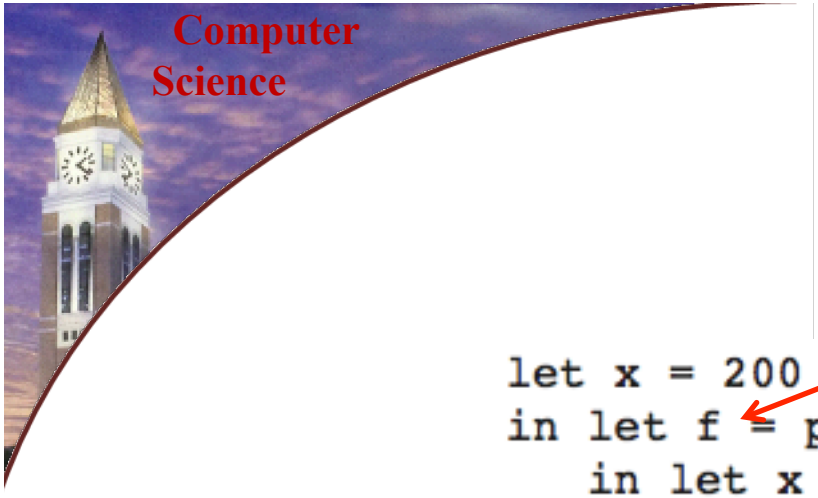

```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```




```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

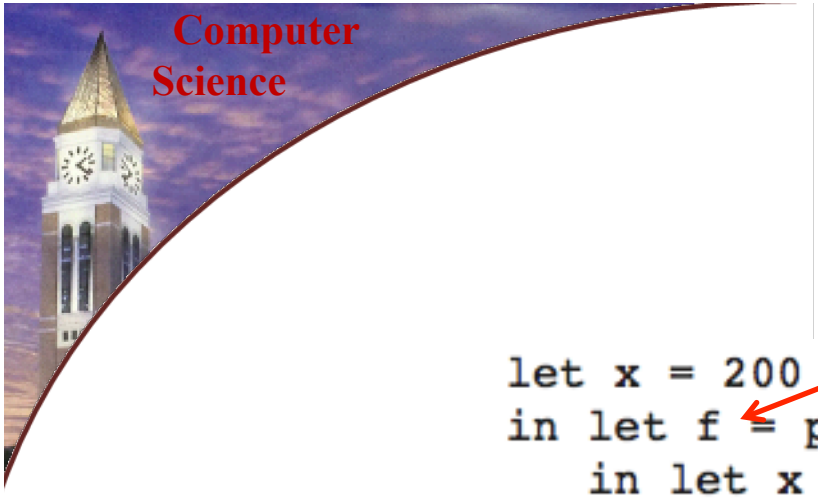



```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```



creation-time
for procedure `f`

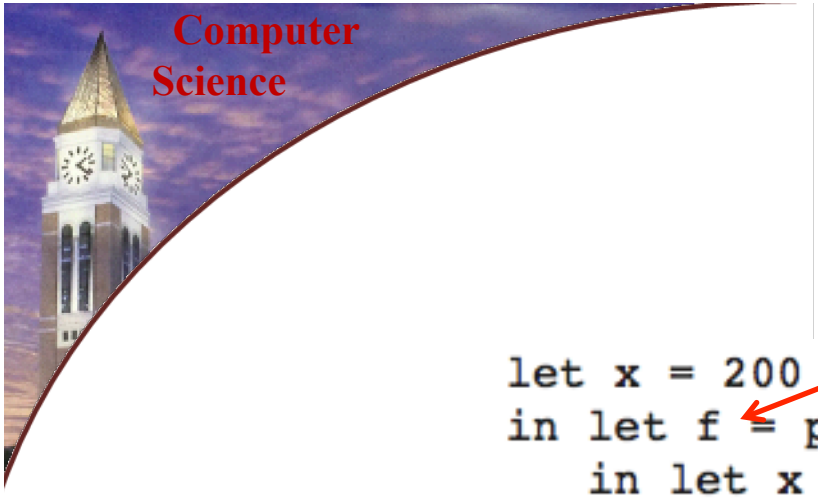
```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```




```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

creation-time for procedure `f`

calling-time for procedure `f`




```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

creation-time for procedure `f`

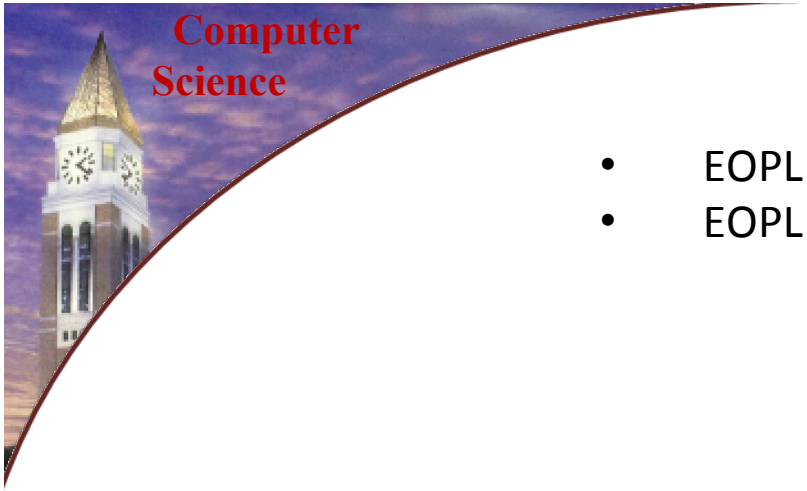
calling-time for procedure `f`

Static Scoping Vs. Dynamic Scoping

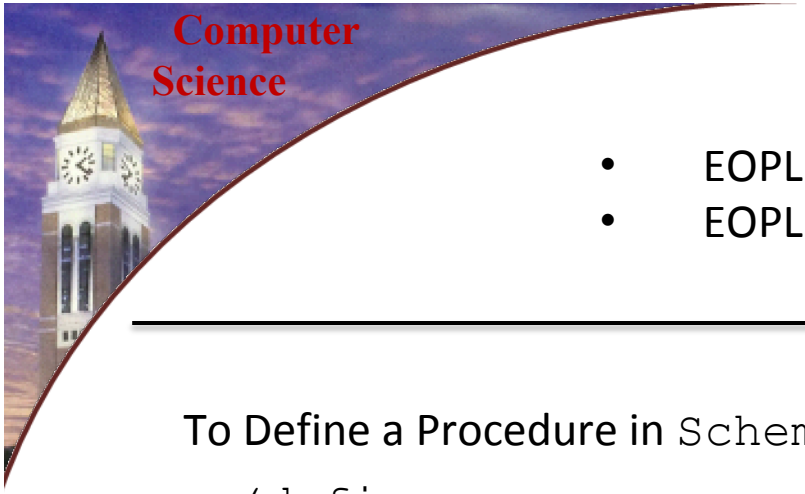
Examples

```
let f = proc (x) -(x,11)
        in f
```

← How about ?



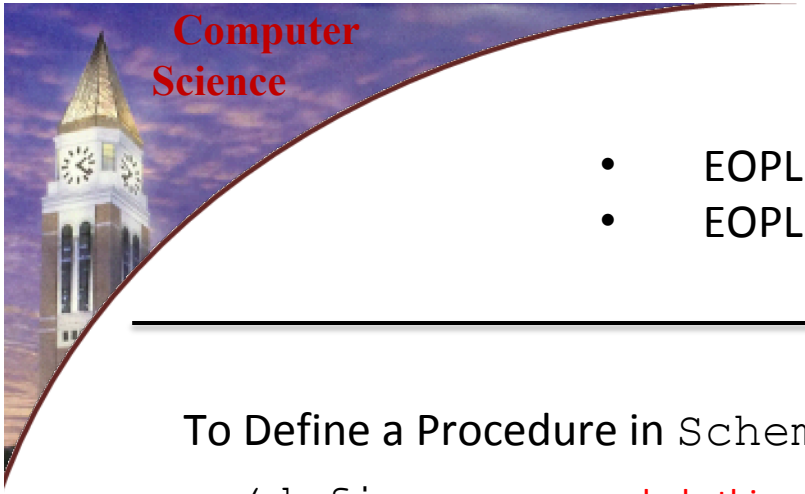
- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)



- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
      (lambda (x) (+ x 1) ) )
```

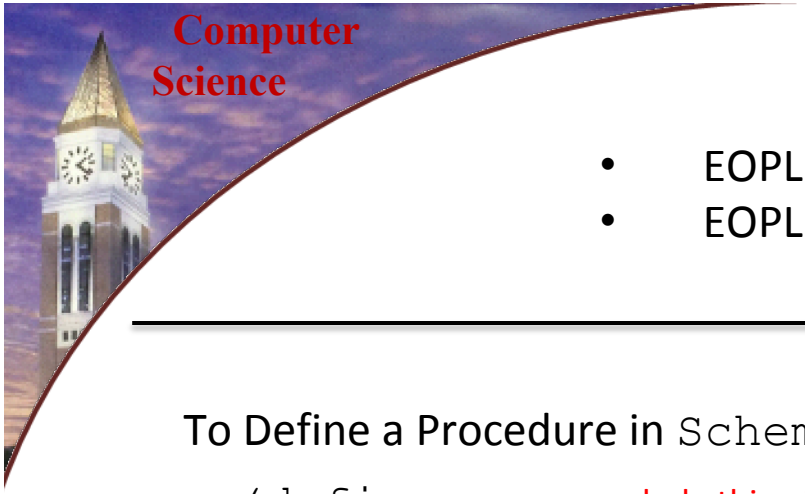


- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
        (lambda (x) (+ x 1) ) )
```

whole thing is one value, given to **a**



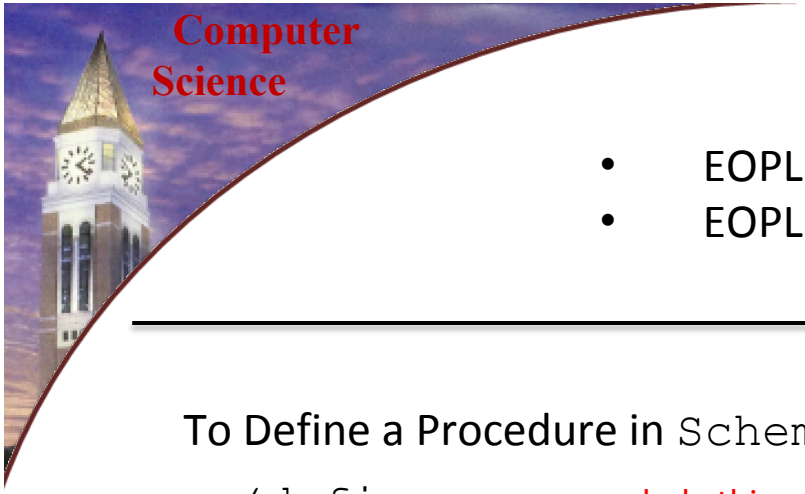
- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
        (lambda (x) (+ x 1) ) )
```

whole thing is one value, given to **a**

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```



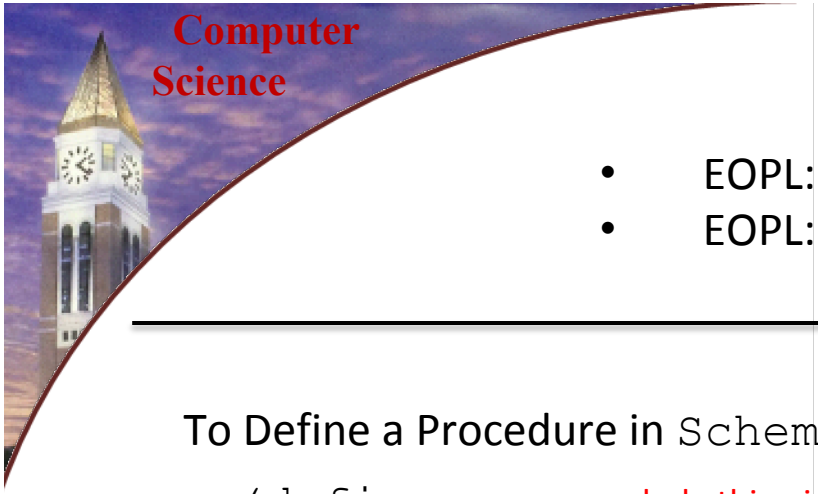
- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
        (lambda (x) (+ x 1) ) )
```

whole thing is one value, given to **a**

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```



- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in Scheme –

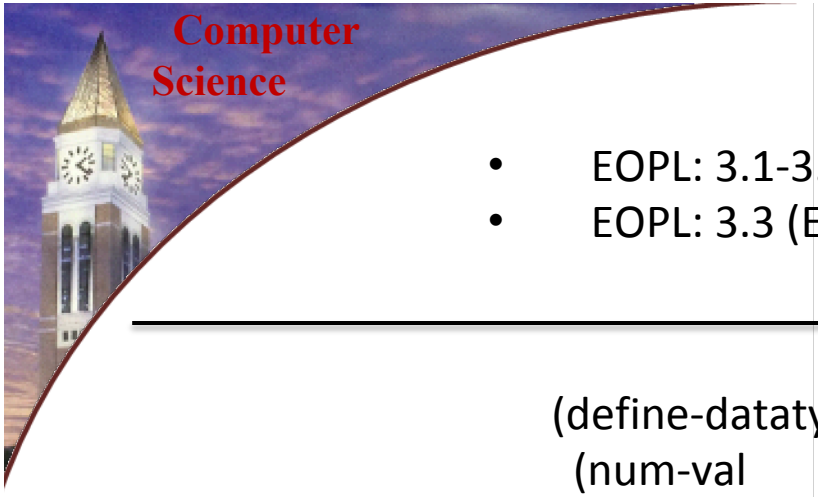
```
(define a
      (lambda (x) (+ x 1) ) )
```

whole thing is one value, given to **a**

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

```
(define-datatype proc proc?
   (procedure
        (var identifier?)
        (body expression?)
        (saved-env environment?)))
```

Found at procedure creation-time, **not** procedure call-time !!



- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

```
(define-datatype proc proc?
   (procedure
       (var identifier?)
       (body expression?)
       (saved-env environment?)))
```

The Extended LET language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Expression ::= `proc (`*Identifier*`)` *Expression*
`proc-exp (var body)`

Expression ::= `(`*Expression* *Expression*`)`
`call-exp (rator rand)`

Figure 3.2 Syntax for the LET language

The `proc` language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Expression ::= `proc (`*Identifier*`)` *Expression*
`proc-exp (var body)`

Expression ::= `(`*Expression* *Expression*`)`
`call-exp (rator rand)`

Figure 3.2 Syntax for the LET language