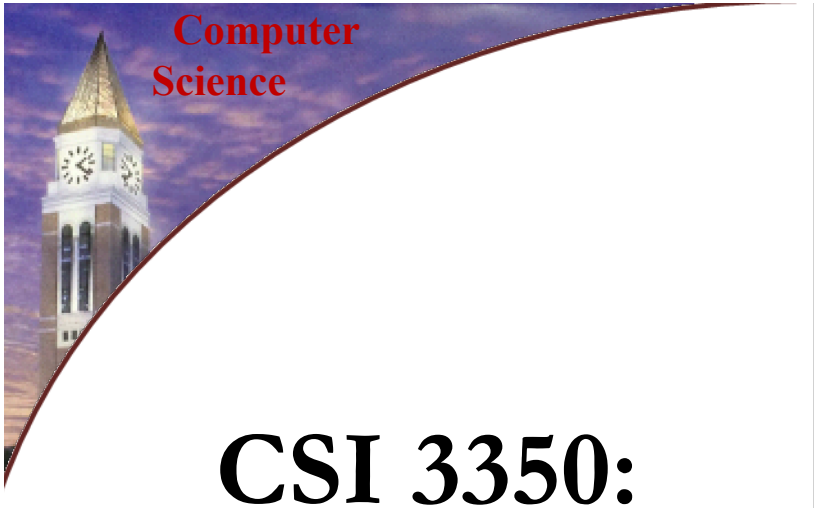


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

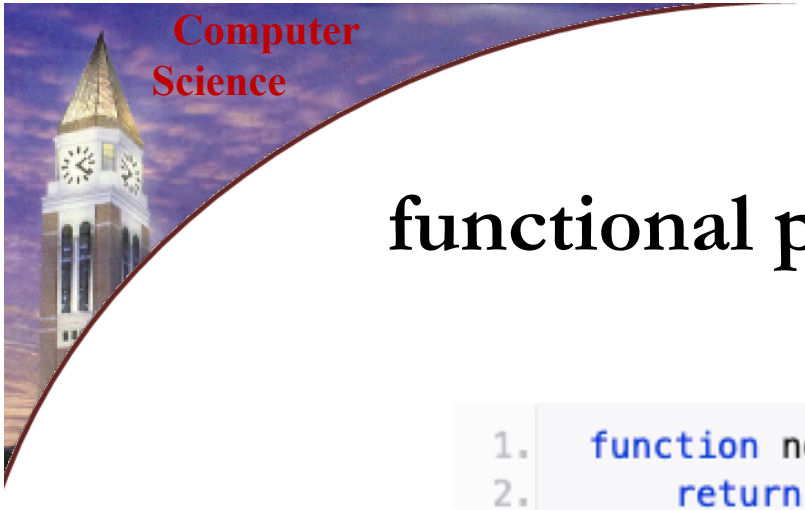
Oakland University

JavaScript, Python are the Most popular programming languages now !

functional programming in JavaScript, Python !

JavaScript, Python are the Most popular programming languages now !

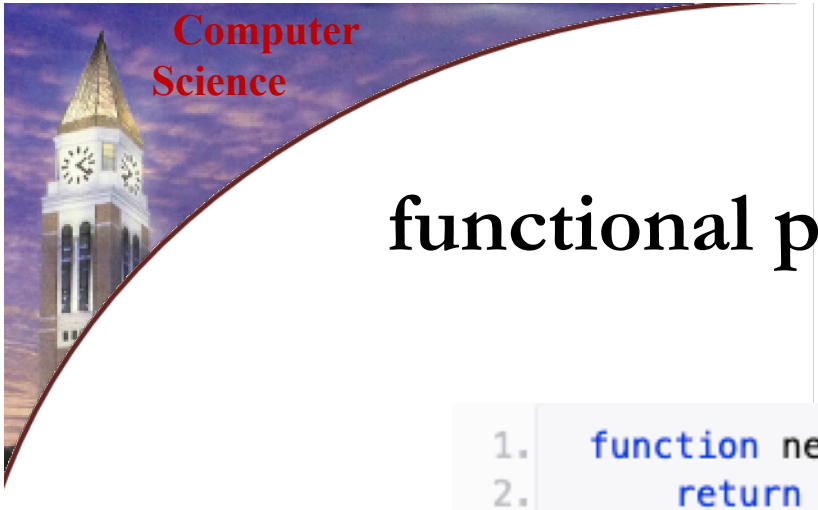
functional programming in JavaScript, Python !



functional programming in JavaScript, Python !

```
1.  function negate (f) {
2.      return function (i) {
3.          return !f(i);
4.    };
5.  }
6.  var isNumber = negate(isNaN);
7.  alert(isNumber(5));
8.  alert(isNumber(NaN));
9.  alert(isNumber("A"));
```

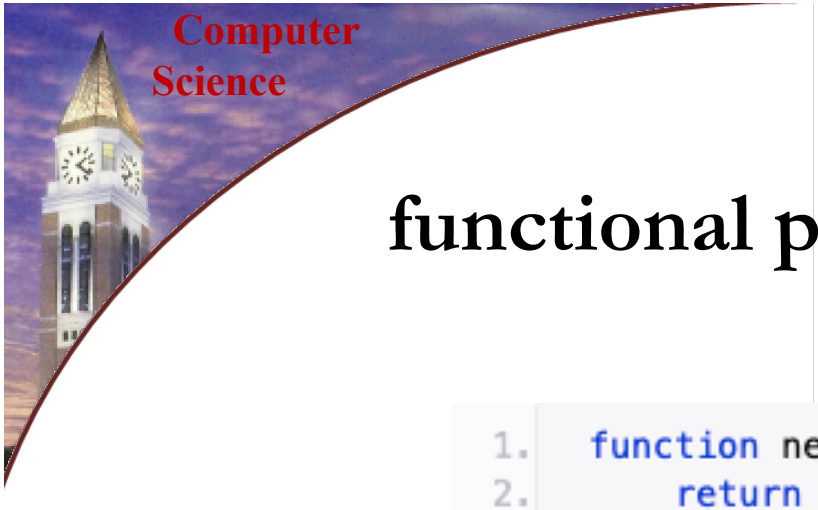
* `alert` function in JavaScript is similar to `System.out.println` in Java



functional programming in JavaScript, Python !

```
1.  function negate (f) {
2.      return function (i) {
3.          return !f(i);
4.    };
5.  }
6.  var isNumber = negate(isNaN);
7.  alert(isNumber(5));
8.  alert(isNumber(NaN));
9.  alert(isNumber("A"));
```

* `alert` function in JavaScript is similar to `System.out.println` in Java
* `isNaN` function in JavaScript returns
  - `true` if its argument **is not** a number,
  - `false` otherwise

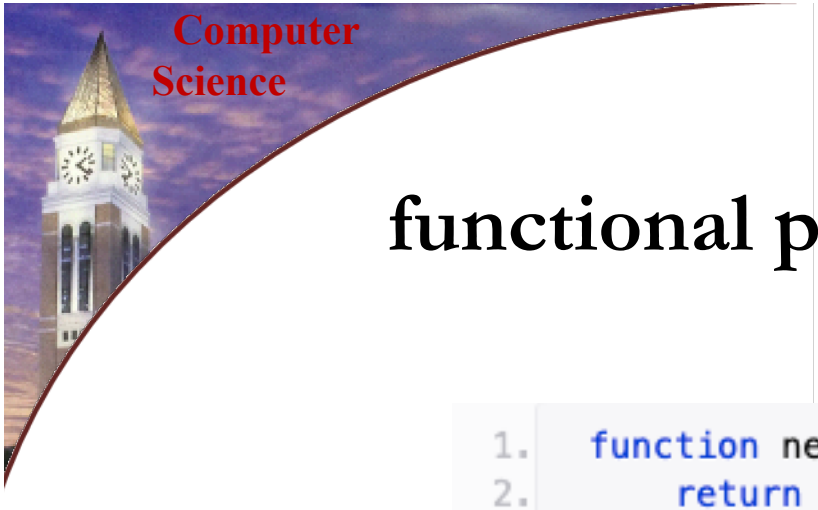


functional programming in JavaScript, Python !

```
1.  function negate (f) {
2.      return function (i) {
3.          return !f(i);
4.    };
5.  }
6.  var isNumber = negate(isNaN);
7.  alert(isNumber(5));
8.  alert(isNumber(NaN));
9.  alert(isNumber("A"));
```

what is the output from line 7 ?

* `alert` function in JavaScript is similar to `System.out.println` in Java
* `isNaN` function in JavaScript returns
  - `true` if its argument is a number,
  - `false` otherwise

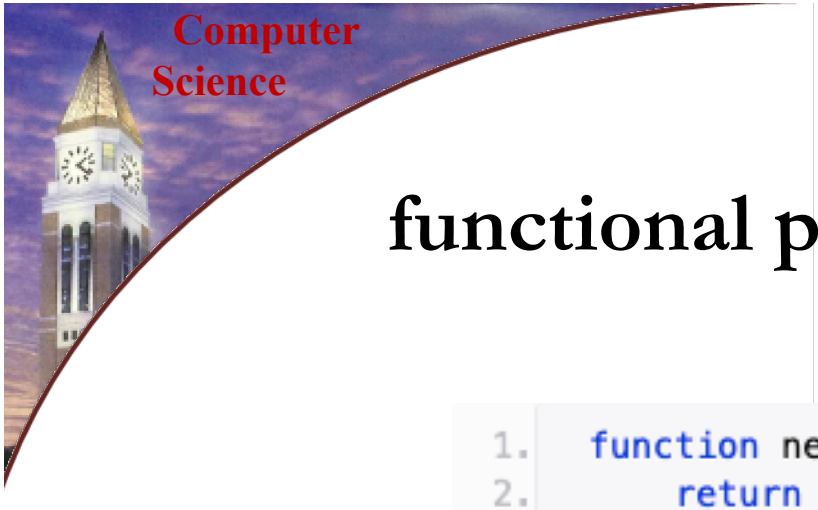


functional programming in JavaScript, Python !

```
1.  function negate (f) {
2.      return function (i) {
3.          return !f(i);
4.    };
5.  }
6.  var isNumber = negate(isNaN);
7.  alert(isNumber(5));
8.  alert(isNumber(NaN));
9.  alert(isNumber("A"));
```

what is the output from line 8 ?

* `alert` function in JavaScript is similar to `System.out.println` in Java
* `isNaN` function in JavaScript returns
  - `true` if its argument is a number,
  - `false` otherwise

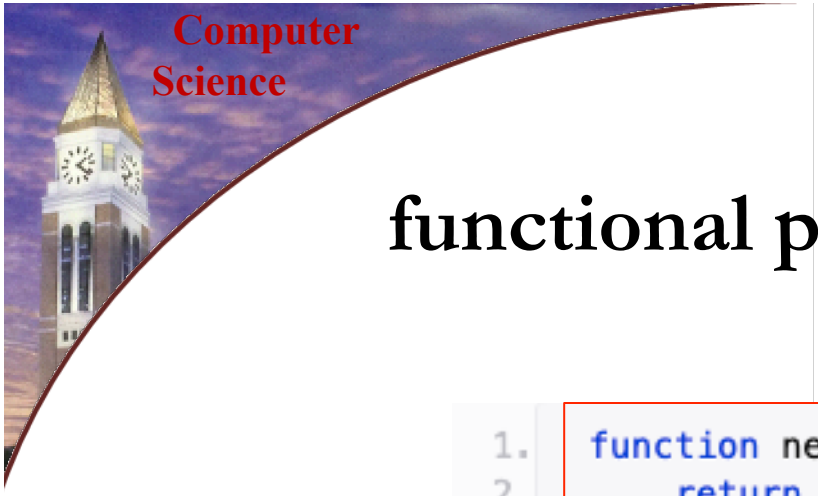


functional programming in JavaScript, Python !

```
1.  function negate (f) {
2.      return function (i) {
3.          return !f(i);
4.    };
5.  }
6.  var isNumber = negate(isNaN);
7.  alert(isNumber(5));
8.  alert(isNumber(NaN));
9.  alert(isNumber("A"));
```

what is the output from line 9 ?

* `alert` function in JavaScript is similar to `System.out.println` in Java
* `isNaN` function in JavaScript returns
  - `true` if its argument is a number,
  - `false` otherwise

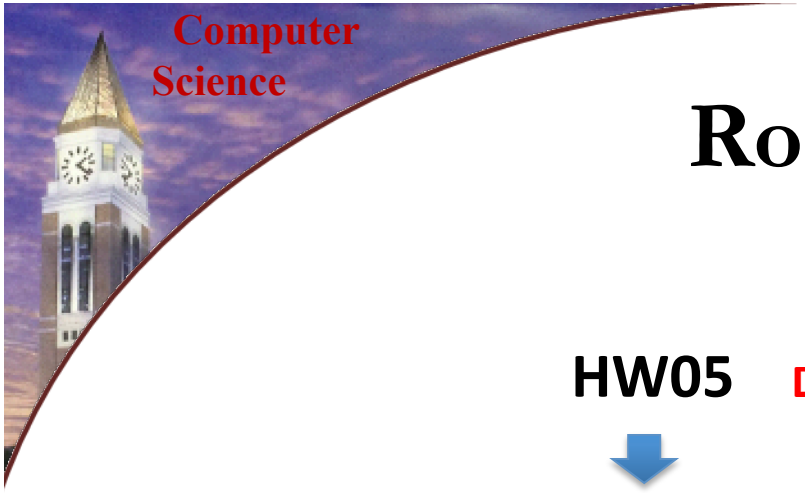


functional programming in JavaScript, Python !

```
1.  function negate (f) {
2.      return function (i) {
3.          return !f(i);
4.    };
5.  }
6.  var isNumber = negate(isNaN);
7.  alert(isNumber(5));
8.  alert(isNumber(NaN));
9.  alert(isNumber("A"));
```

what is the equivalent code in Scheme for `negate` function?

* `alert` function in JavaScript is similar to `System.out.println` in Java
* `isNaN` function in JavaScript returns
  - `true` if its argument is a number,
  - `false` otherwise



Road Ahead -

HW05 Due: Nov 12

↓

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

↓

Exam02 ← **Nov 27**

↓

HW07

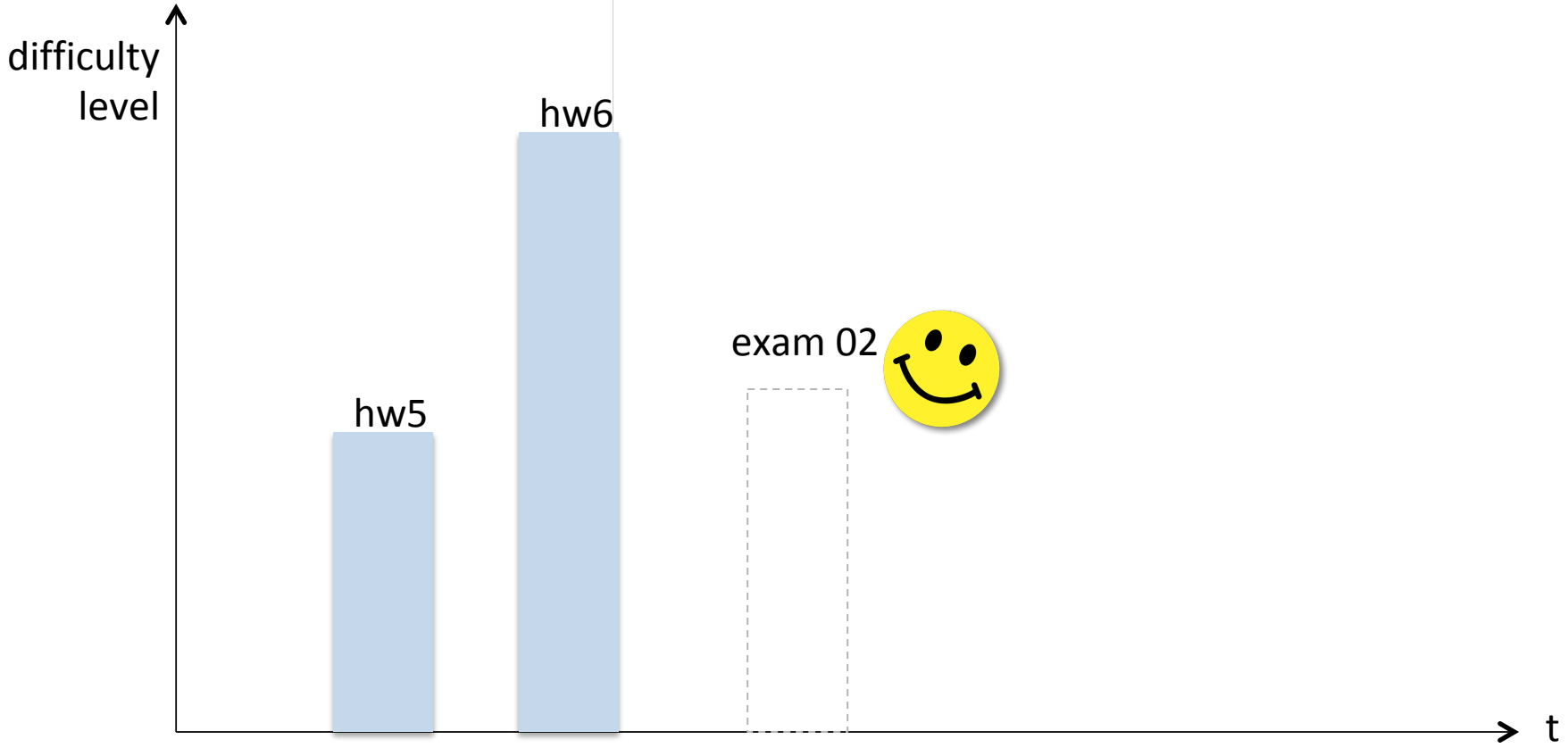
↓

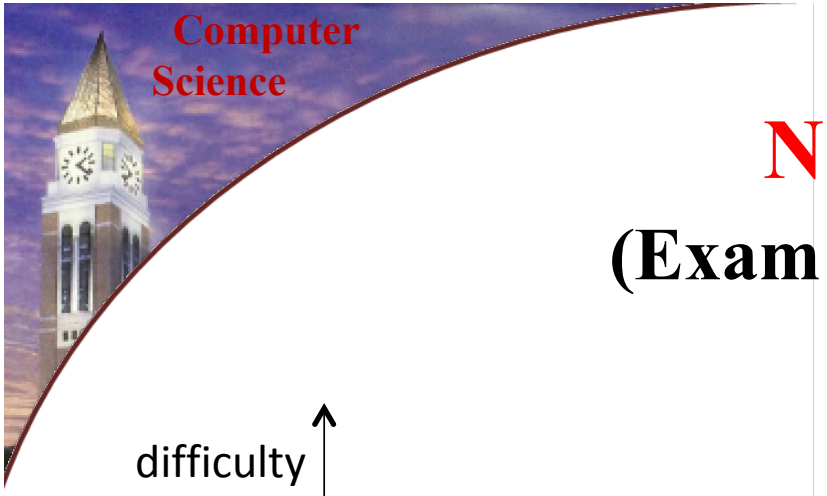
Final Exam : 7pm ~10pm : Dec 09, 2019

Exam 02

Nov 27 (in class)

(Exam 02 covers HW 5 ~ 6)

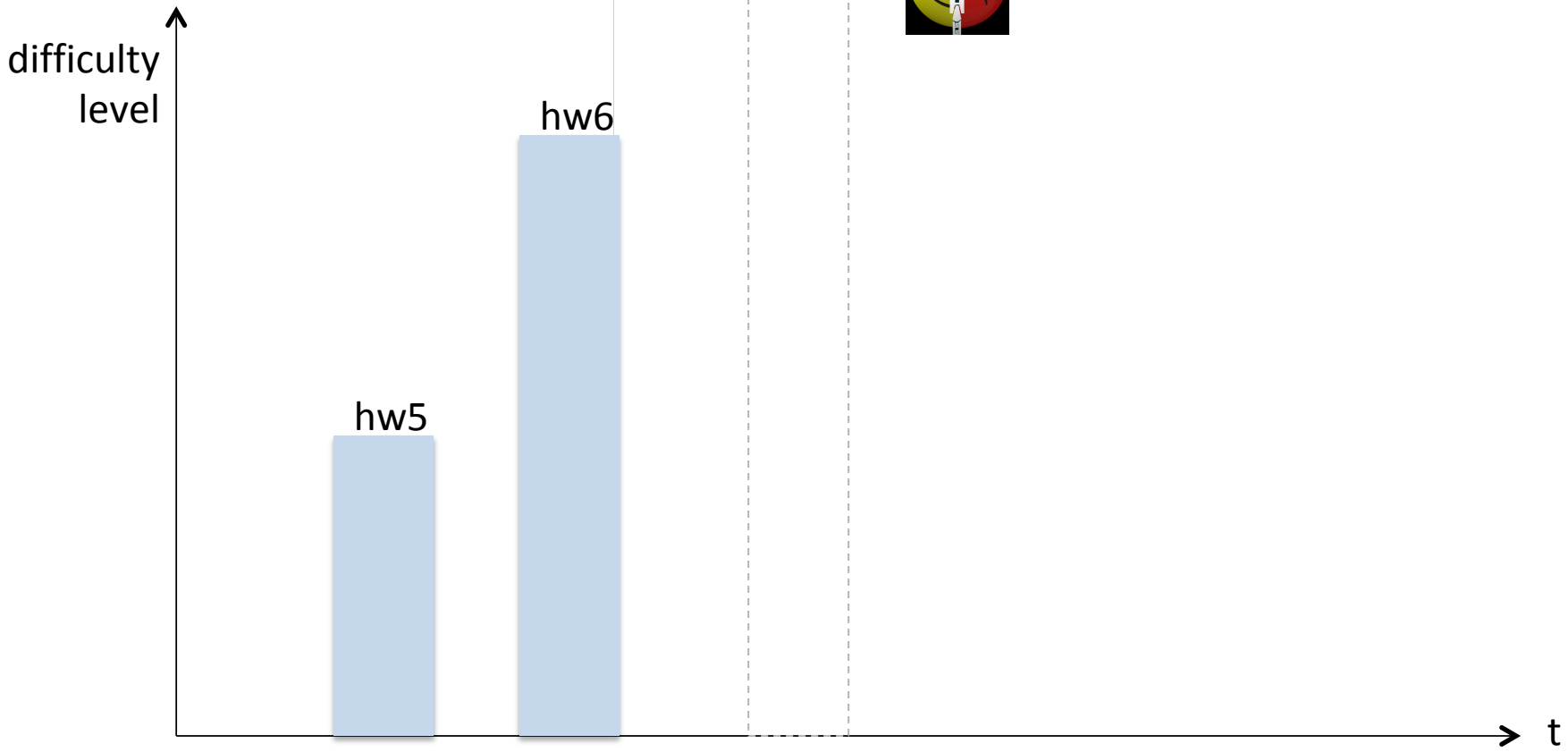




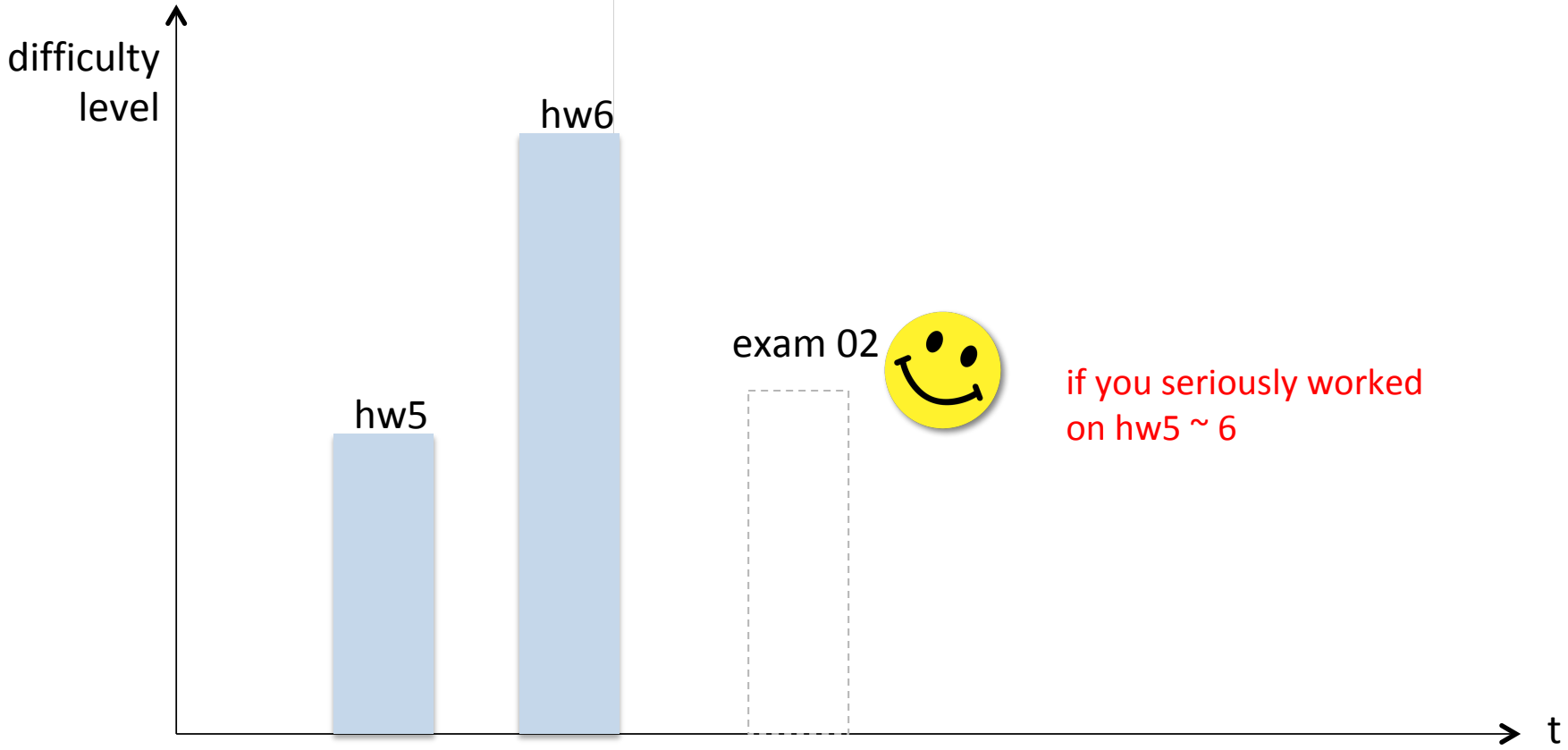
Exam 02

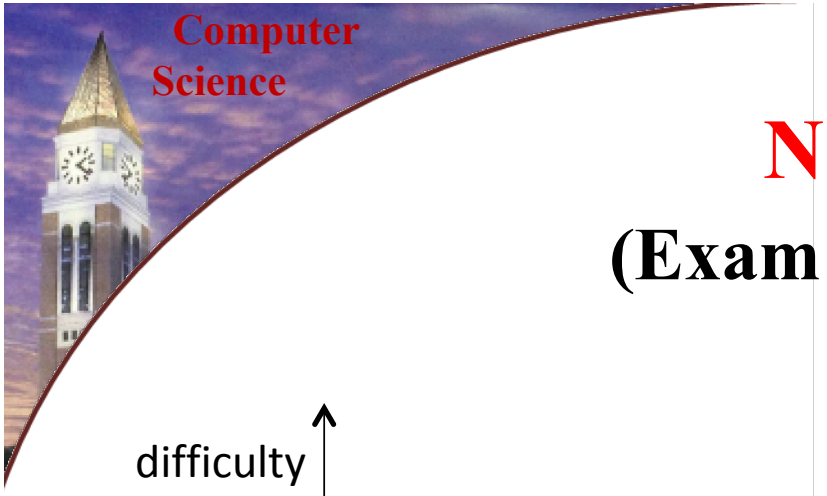
Nov 27 (in class)

(Exam 02 covers HW 5 ~ 6)



Exam 02
Nov 27 (in class)
(Exam 02 covers HW 5 ~ 6)





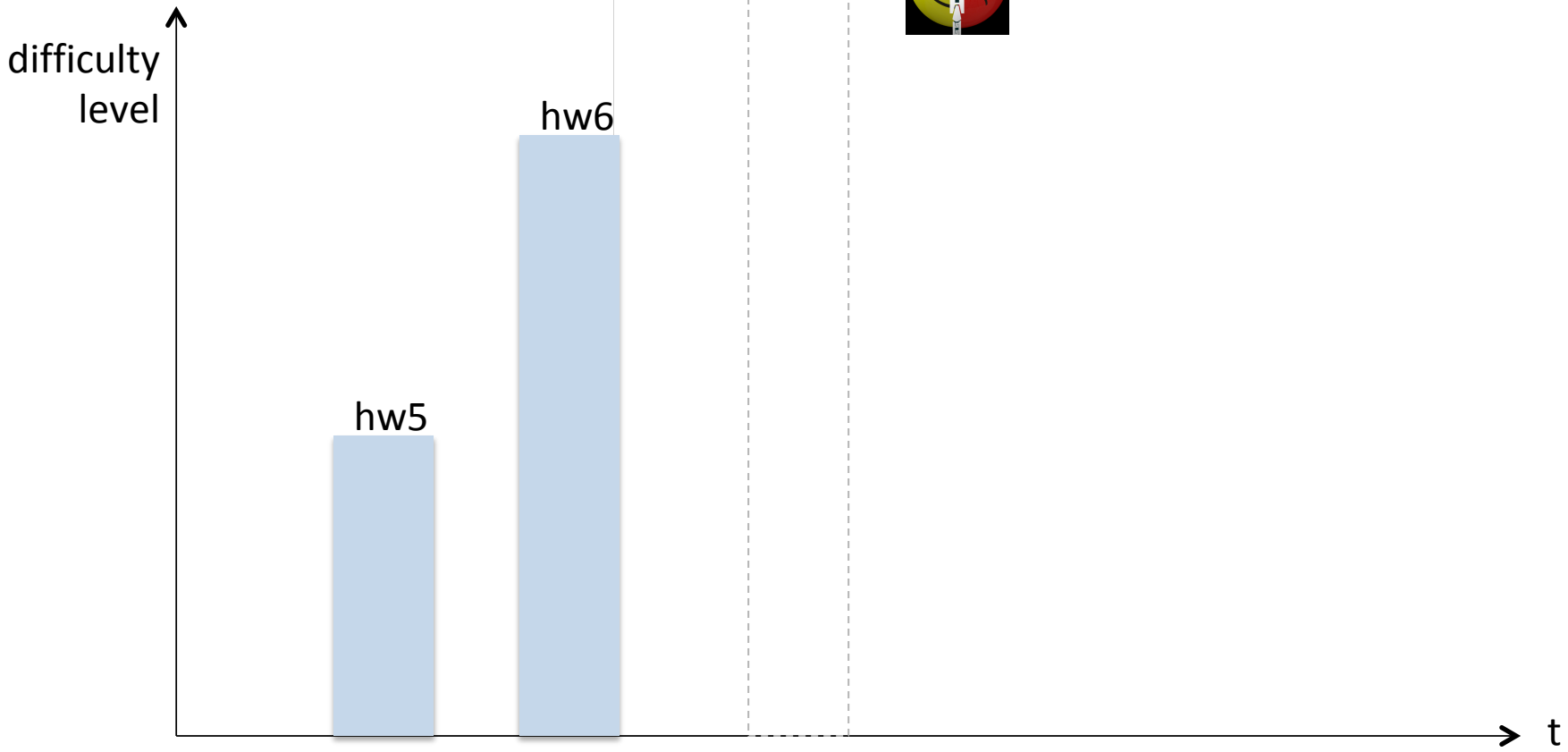
Exam 02

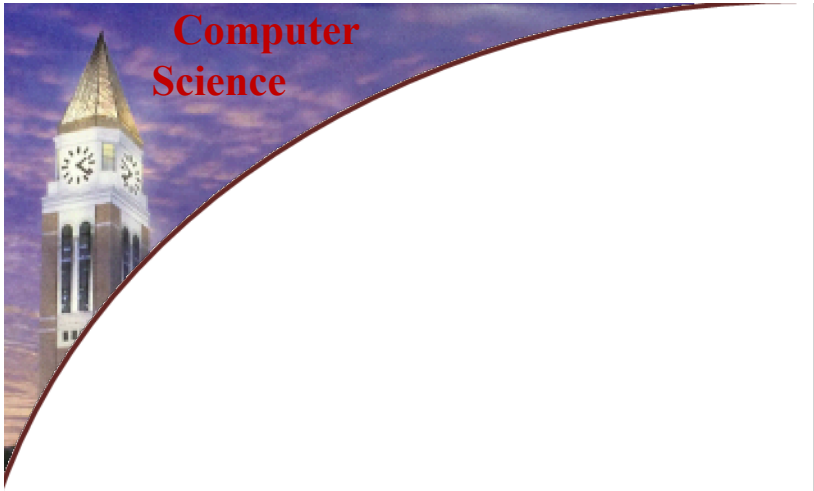
Nov 27 (in class)

(Exam 02 covers HW 5 ~ 6)

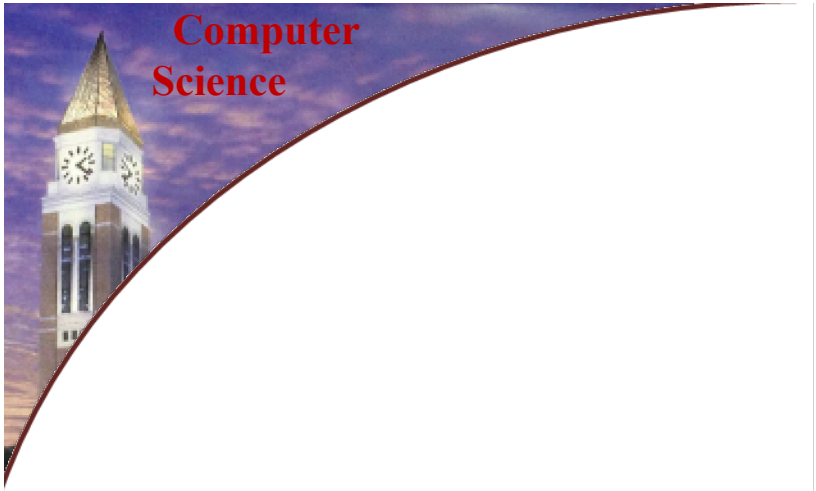
exam 02

otherwise …





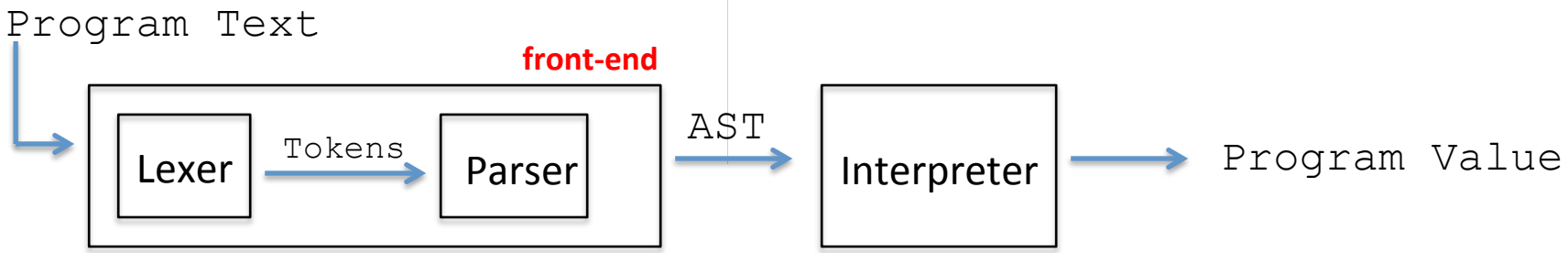
IMPLEMENTING A PROGRAMMING LANGUAGE OF YOUR DESIGN



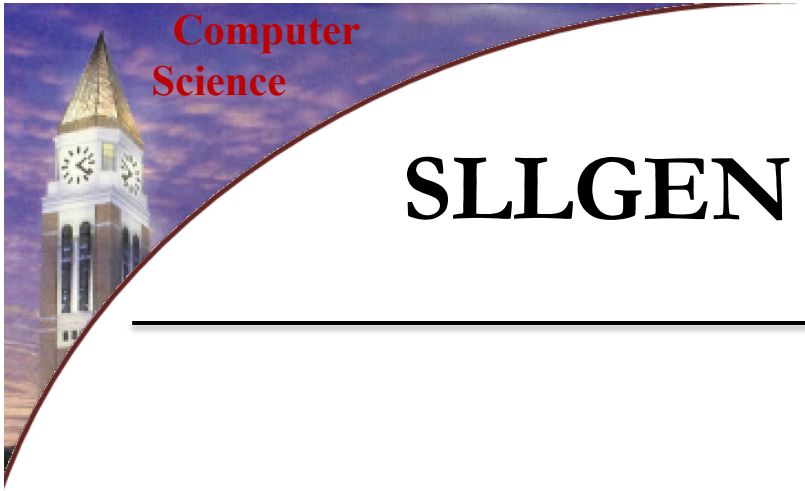
Suggested reading:

- EOPL: 2.4 (refresh your memory on define-datatype)
- EOPL: B.1-B.3 (about sllgen)
- EOPL: 3.1-3.2 (implementation of LET language)

Structural Overview

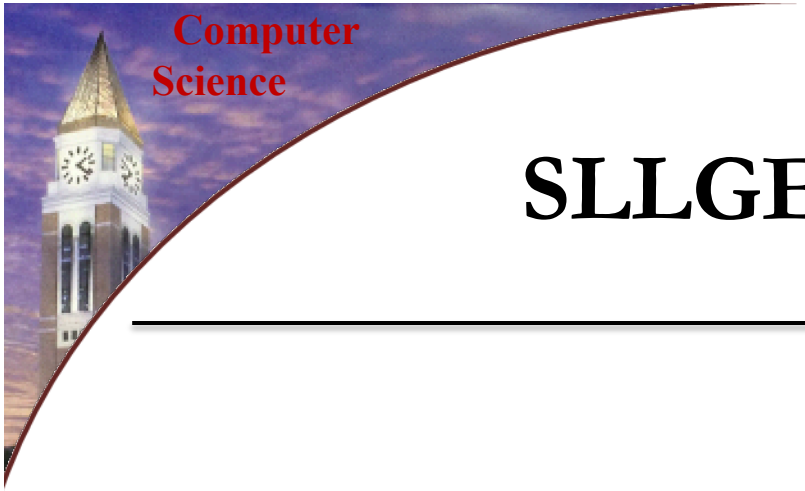


AST: **A**bstract **S**yntax **T**ree



SLLGEN Boiler Plate Code

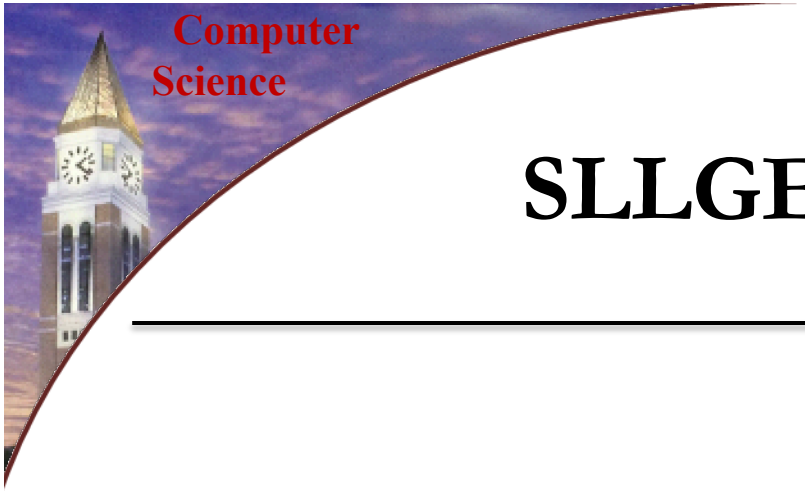
```
(sllgen:make-define-datatypes lexical-spec grammar-spec)

(define (show-data-types)
  (sllgen:list-define-datatypes lexical-spec grammar-spec))


(define parser
  (sllgen:make-string-parser lexical-spec grammar-spec))

(define scanner
  (sllgen:make-string-scanner lexical-spec grammar-spec))
```



SLLGEN Boiler Plate

```
(define (show-data-types)
  (sllgen:list-define-datatypes lexical-spec grammar-spec))
```

define the Abstract Syntax Tree for the **grammar** as the return value of **(show-data-types)** function.

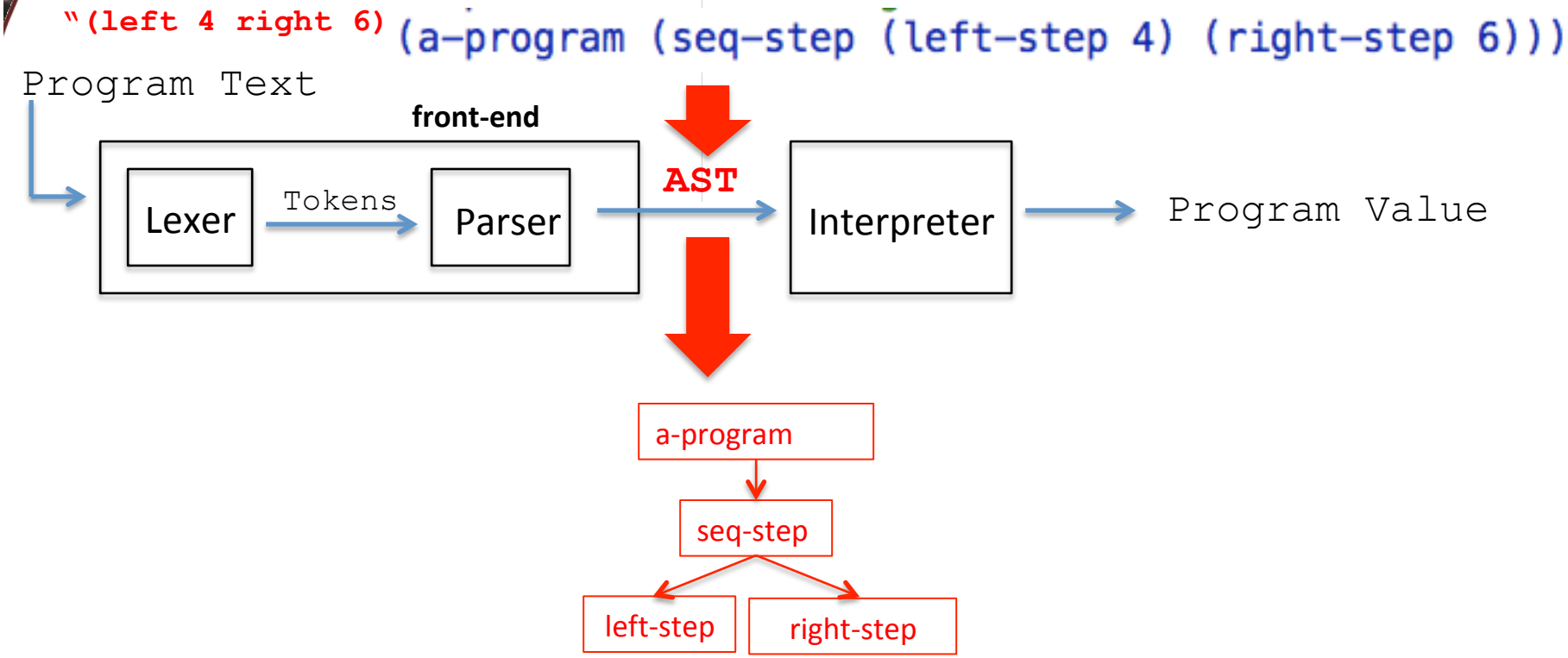


SLLGEN Boiler Plate

`parser` is a **one argument function** that takes a program string, scans & parses it and generates the corresponding **Abstract Syntax Tree**.

```
(define parser
  (sllgen:make-string-parser lexical-spec grammar-spec))
```

Internal Representation of Program Values



Grammar For LET Language (EOPL p60)

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)`

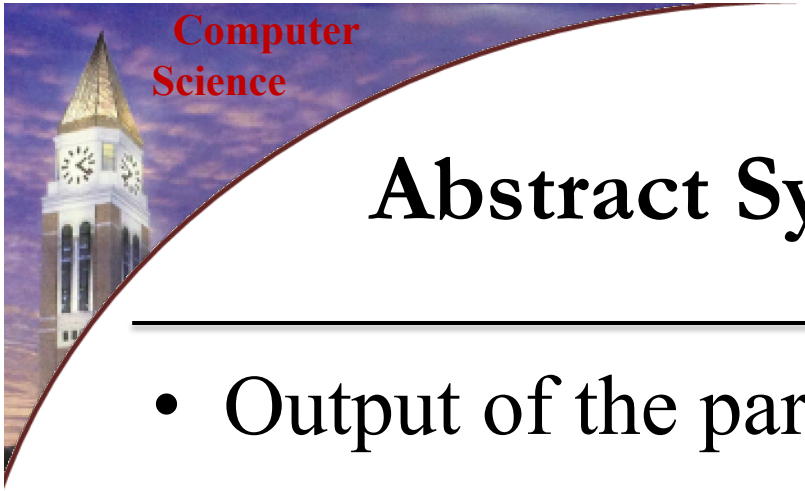
Expression ::= *Identifier*
`var-exp (var)`

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

What does a parser return to us?

Concrete Syntax

Abstract Syntax



Abstract Syntax: An Example

- Output of the parser: an AST

Concrete syntax for if-expression

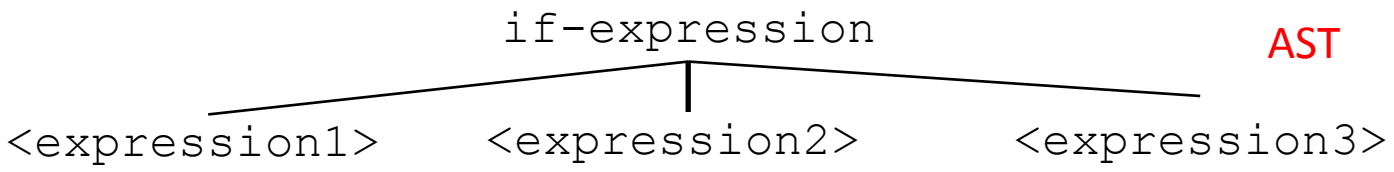
`if` <expression> `then` <expression> `else` <expression>

The Abstract Syntax

Concrete syntax for if-expression

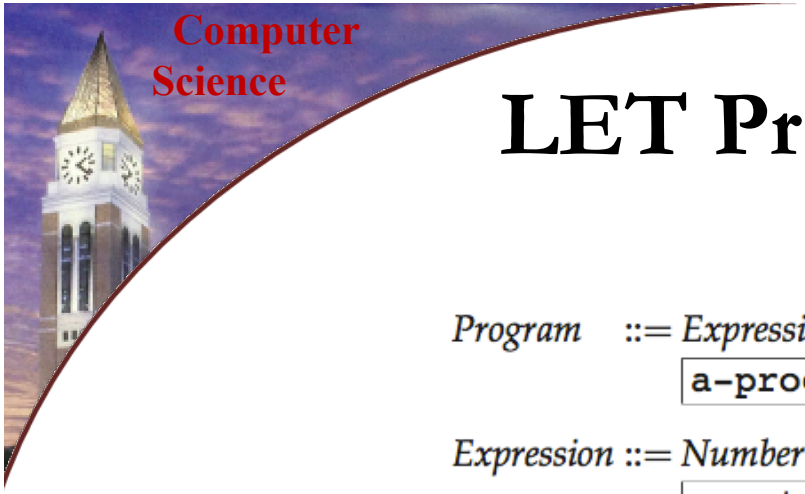
`if` <expression1> `then` <expression2> `else` <expression3>

The essential structure of `if-expression`



linearize AST

```
if-expression (exp1 exp2 exp3)
```



LET Programming Language (EOPL p60)

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

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

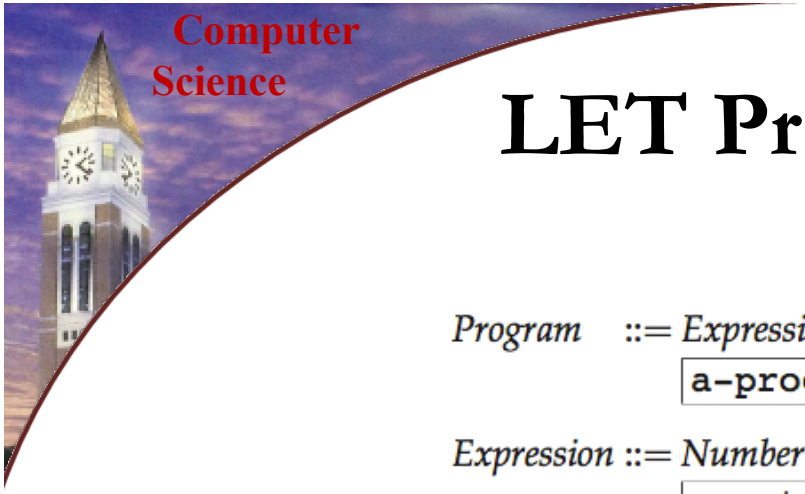
Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

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)`



LET Programming Language (EOPL p60)

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

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

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

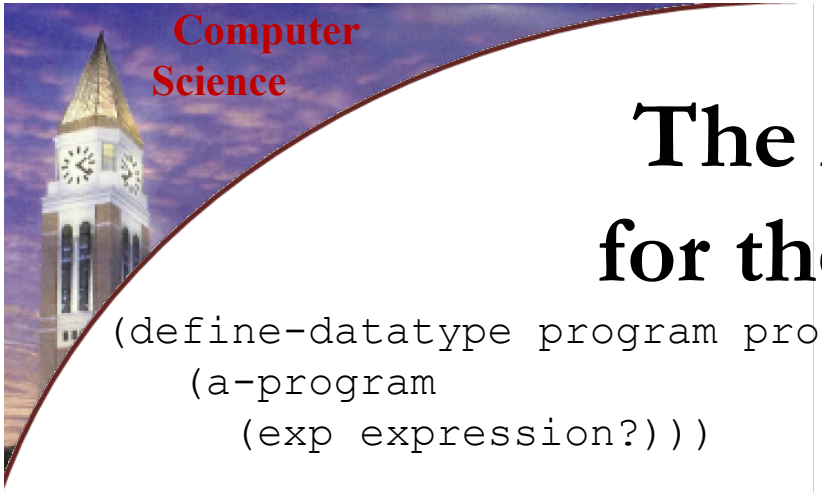
Expression ::= `zero?` `(` *Expression* `)`
`zero?-exp (exp1)`

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)`

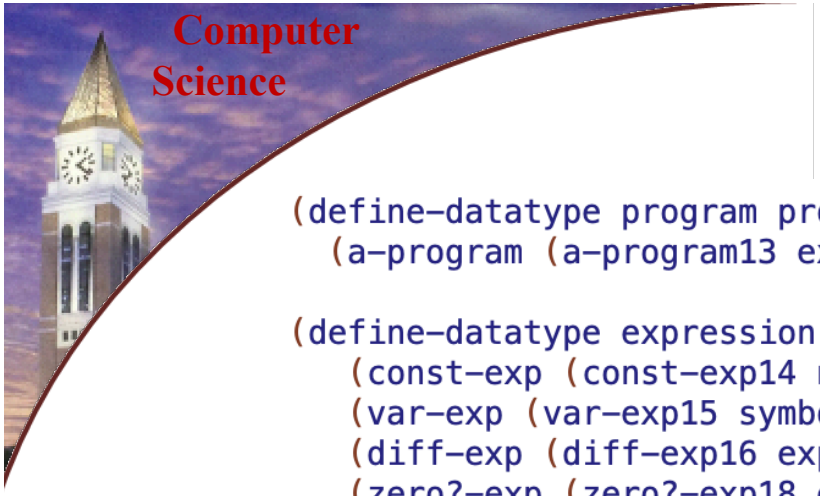
An implementation of the interpreter for **`LET`** programming language is given in EOPL starting from p.60



The Abstract Syntax for the LET Language

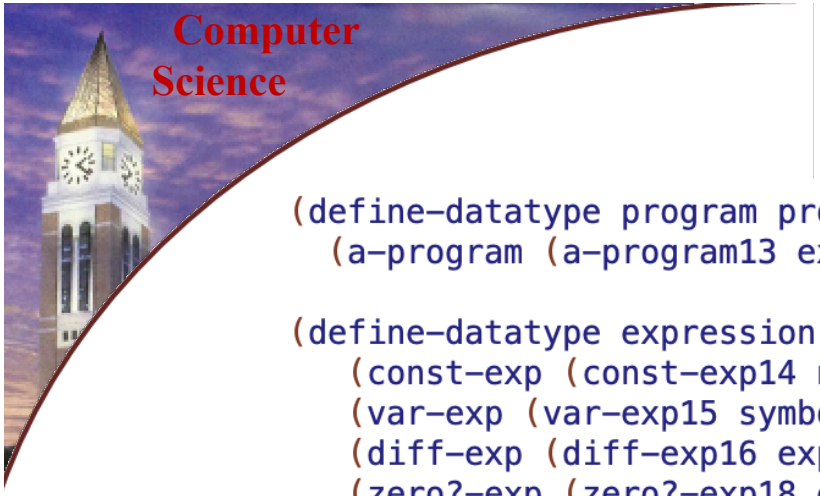
```
(define-datatype program program?
   (a-program
     (exp expression?)))

(define-datatype expression expression?
   (const-exp
     (num number?))
   (var-exp
     (var symbol?))
   (diff-exp
     (exp1 expression?) (exp2 expression?))
   (zero?-exp
     (exp expression?))
   (if-exp
     (exp1 expression?) (exp2 expression?) (exp3 expression?))
   (let-exp
     (var symbol?) (val-exp expression?) (body expression?))
)
```

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




```
(define-datatype program program?
  (a-program (a-program13 expression?)))

(define-datatype expression expression?
   (const-exp (const-exp14 number?))
   (var-exp (var-exp15 symbol?))
   (diff-exp (diff-exp16 expression?) (diff-exp17 expression?))
   (zero?-exp (zero?-exp18 expression?))
   (if-exp (if-exp19 expression?) (if-exp20 expression?) (if-exp21 expression?))
   (let-exp (let-exp22 symbol?) (let-exp23 expression?) (let-exp24 expression?)))
```

```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comment ("#" (arbno (not #\newline))) skip)
    (num (digit (arbno digit)) number)
    (identifier (letter (arbno (or letter digit "_" "-" "?"))) symbol)))


(define grammar-spec
  '(
    (program (expression) a-program)
    (expression (num) const-exp)
    (expression (identifier) var-exp)
    (expression ("-" "(" expression "," expression ")" ) diff-exp)
    (expression ("zero?" "(" expression ")") zero?-exp)
    (expression ("if" expression "then" expression "else" expression) if-exp)
    (expression ("let" identifier "=" expression "in" expression) let-exp)))
```

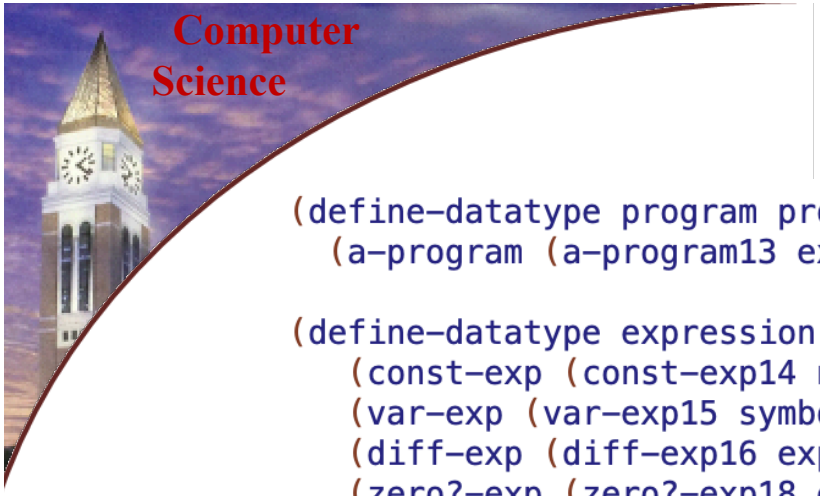


abstract syntax

```
(define-datatype program program?
  (a-program (a-program13 expression?)))

(define-datatype expression expression?
   (const-exp (const-exp14 number?))
   (var-exp (var-exp15 symbol?))
   (diff-exp (diff-exp16 expression?) (diff-exp17 expression?))
   (zero?-exp (zero?-exp18 expression?))
   (if-exp (if-exp19 expression?) (if-exp20 expression?) (if-exp21 expression?))
   (let-exp (let-exp22 symbol?) (let-exp23 expression?) (let-exp24 expression?)))
```

```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comment ("#" (arbno (not #\newline))) skip)
    (num (digit (arbno digit)) number)
    (identifier (letter (arbno (or letter digit "_" "-" "?"))) symbol)))


(define grammar-spec
  '(
    (program (expression) a-program)
    (expression (num) const-exp)
    (expression (identifier) var-exp)
    (expression ("-" "(" expression "," expression ")" ) diff-exp)
    (expression ("zero?" "(" expression ")") zero?-exp)
    (expression ("if" expression "then" expression "else" expression) if-exp)
    (expression ("let" identifier "=" expression "in" expression) let-exp)))
```

concrete syntax



abstract syntax

```
(define-datatype program program?
  (a-program (a-program13 expression?)))

(define-datatype expression expression?
   (const-exp (const-exp14 number?))
   (var-exp (var-exp15 symbol?))
   (diff-exp (diff-exp16 expression?) (diff-exp17 expression?))
   (zero?-exp (zero?-exp18 expression?))
   (if-exp (if-exp19 expression?) (if-exp20 expression?) (if-exp21 expression?))
   (let-exp (let-exp22 symbol?) (let-exp23 expression?) (let-exp24 expression?)))
```

(show-data-types)

```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comment ("#" (arbno (not #\newline))) skip)
    (num (digit (arbno digit)) number)
    (identifier (letter (arbno (or letter digit "_" "-" "?"))) symbol)))


(define grammar-spec
  '(
    (program (expression) a-program)
    (expression (num) const-exp)
    (expression (identifier) var-exp)
    (expression ("-" "(" expression "," expression ")" ) diff-exp)
    (expression ("zero?" "(" expression ")") zero?-exp)
    (expression ("if" expression "then" expression "else" expression) if-exp)
    (expression ("let" identifier "=" expression "in" expression) let-exp)))
```

concrete syntax

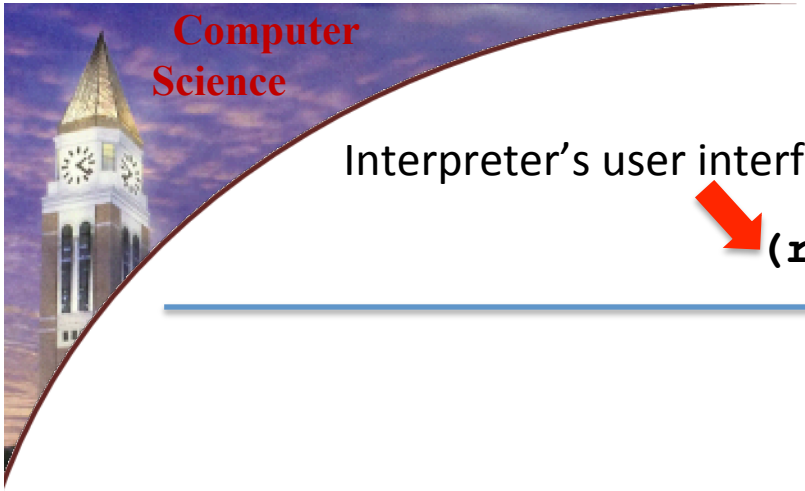
HW06

```
<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

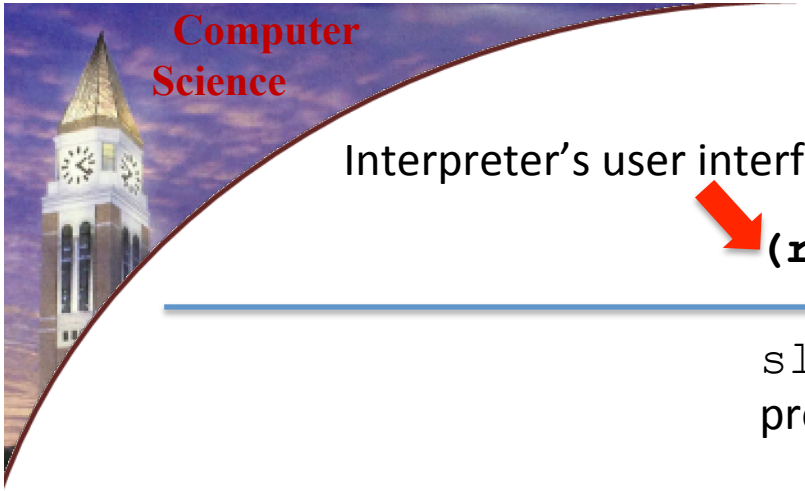
(Problem 3)

```
<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"

          | (<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"
```



Interpreter's user interface

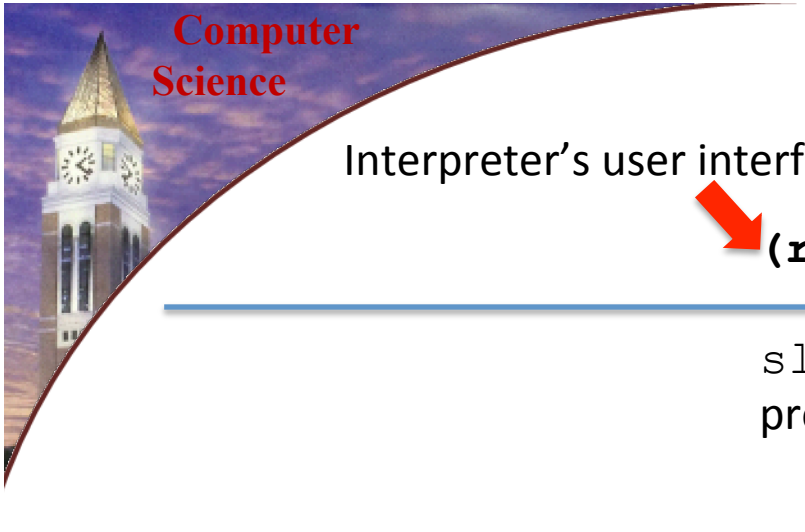
```
(run program-string)
```



Interpreter's user interface

```
(run program-string)
```

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

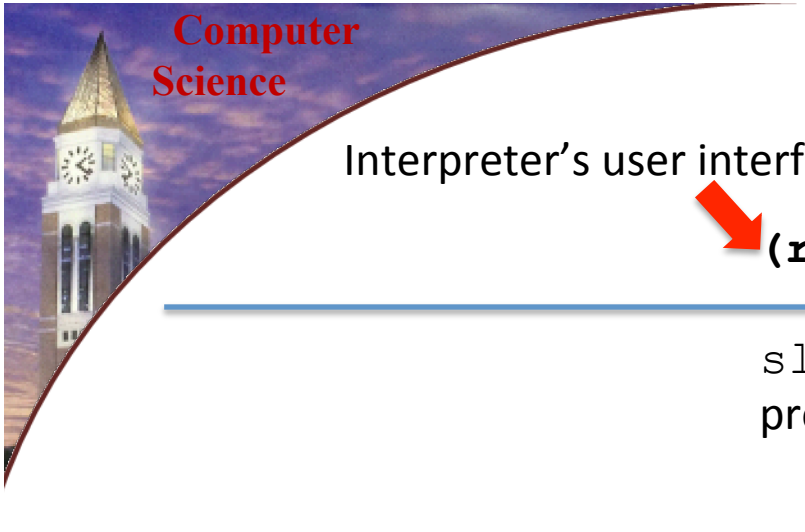


Interpreter's user interface

`(run program-string)`

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

`ast` = (`parser` programing-string)



Interpreter's user interface

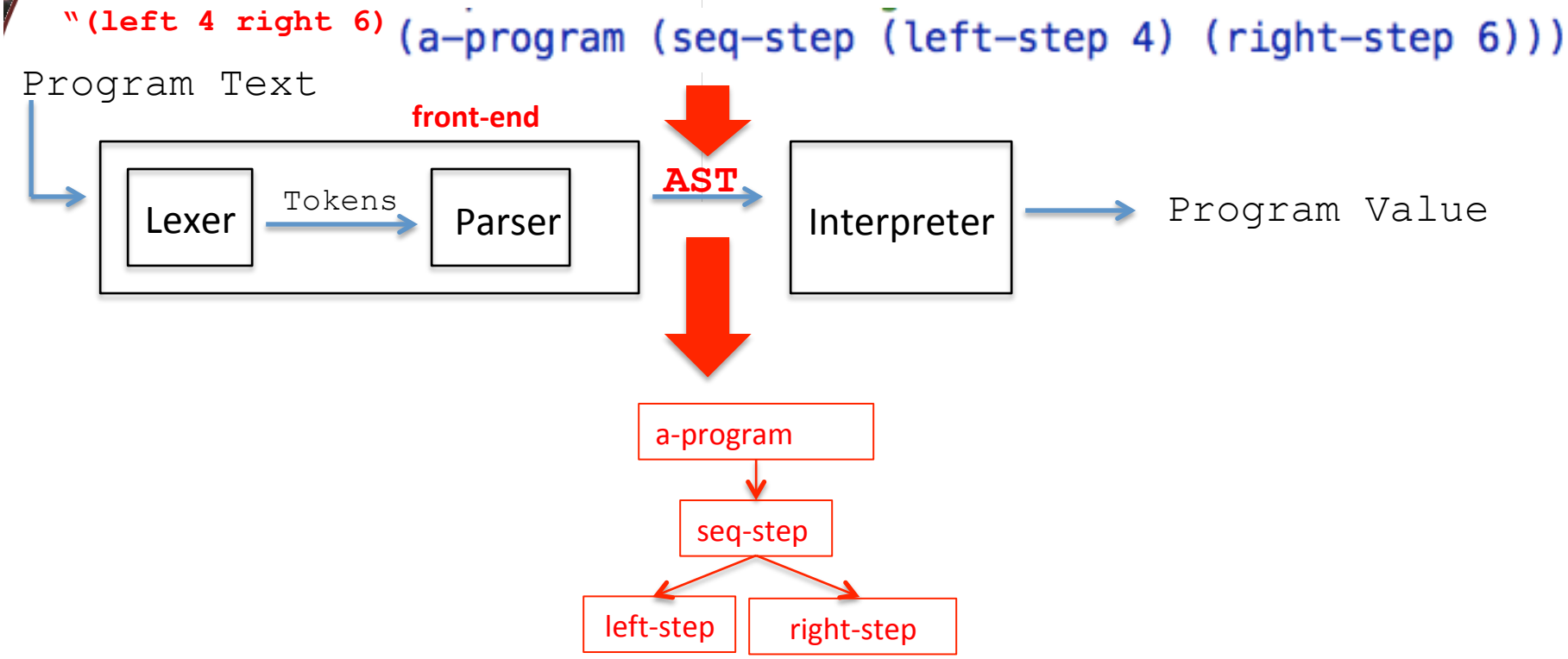
`(run program-string)`

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

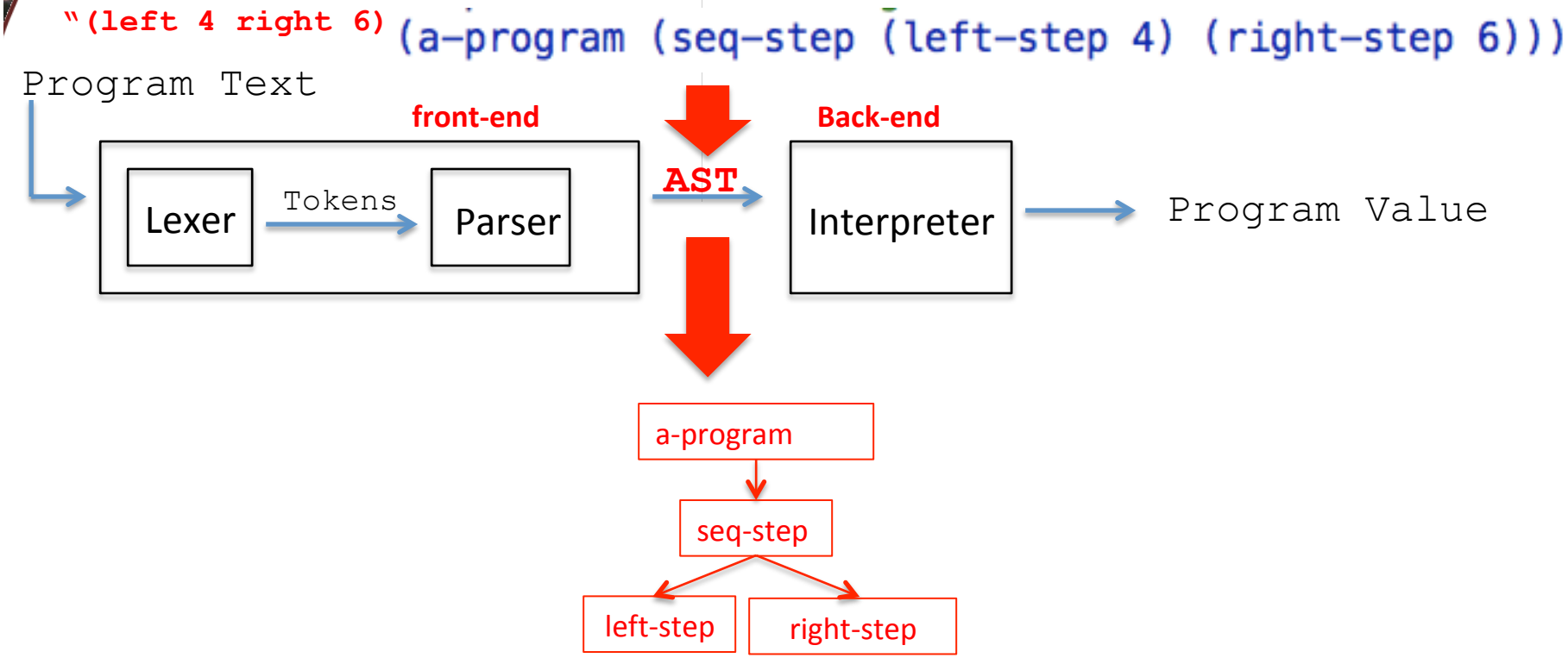
`ast` = (`parser` programing-string)

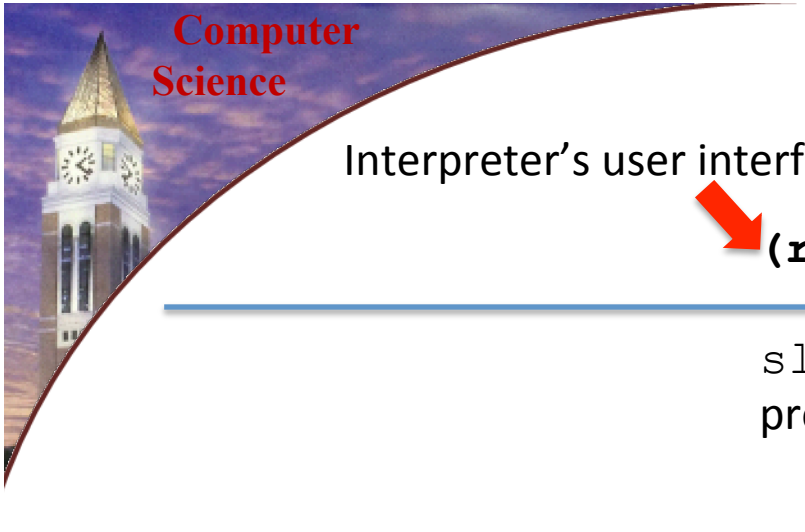
Back-end

Internal Representation of Program Values



Internal Representation of Program Values





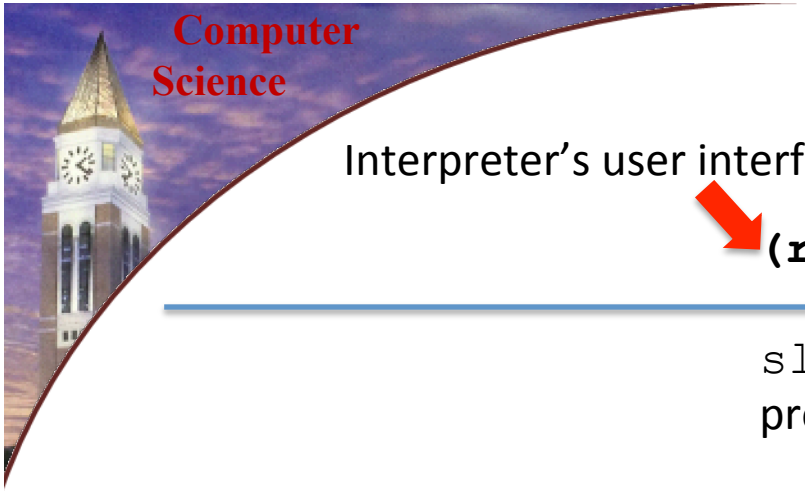
Interpreter's user interface

`(run program-string)`

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

`ast` = (`parser` programing-string)

Back-end



Interpreter's user interface

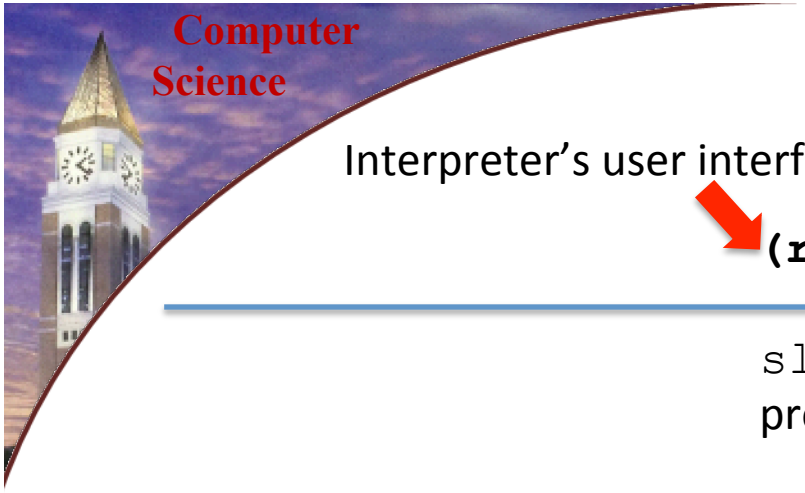
`(run program-string)`

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

`ast` = (`parser` programing-string)

(value-of `ast`)

Back-end



Interpreter's user interface

`(run program-string)`

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

`ast` = (`parser` programing-string)

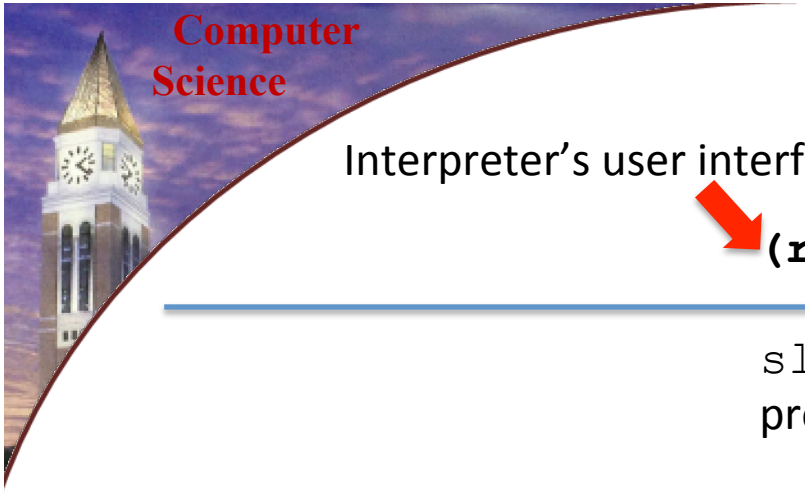
(value-of `ast`)

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

Back-end

(value-of-program `ast`)

(value-of-expr `ast`)



Interpreter's user interface

`(run program-string)`

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

`ast` = (`parser` programing-string)

(value-of `ast`)

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

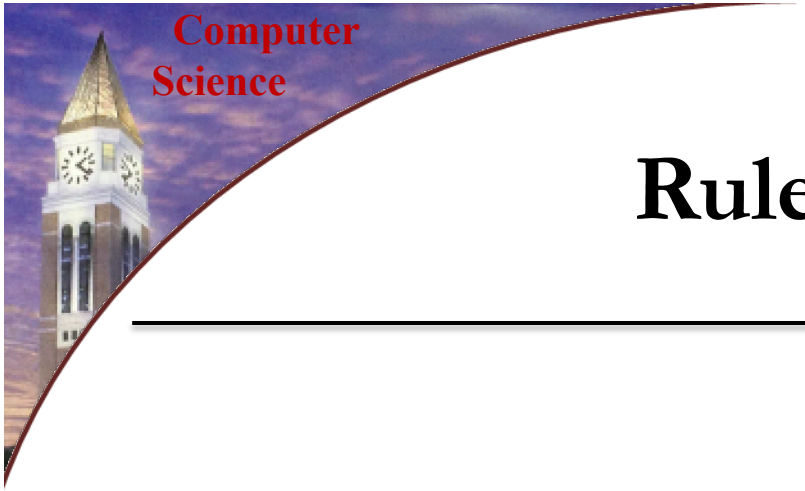
Back-end

(value-of-program `ast`)

(value-of-expr `ast`)

define-datatype:

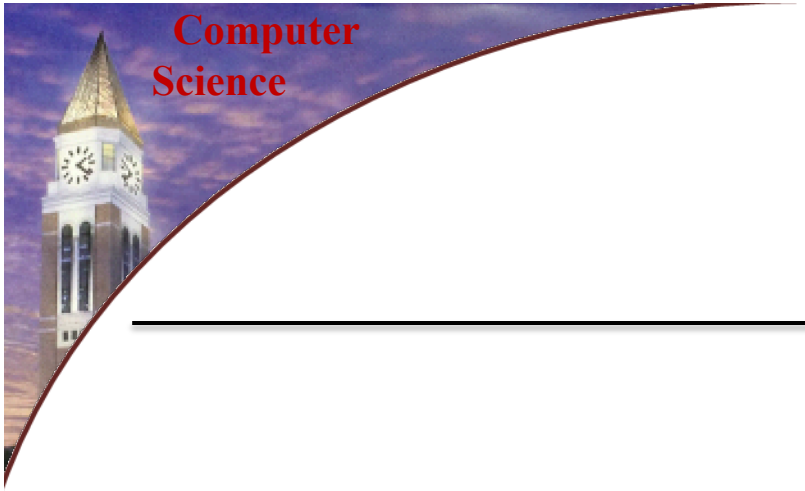
- think of each non-terminal in a grammar production as one data type
- each non-terminal takes different variants (num-expr, str-expr, ... etc.)



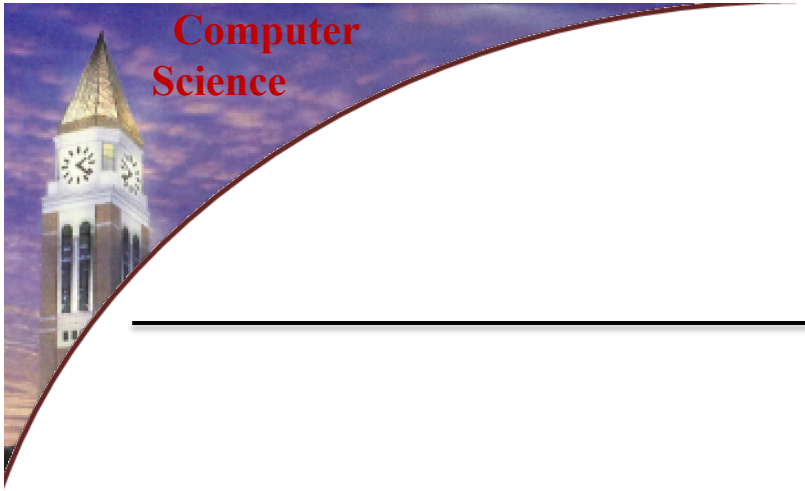
Rule Of Thumb

Every expression will return a value!

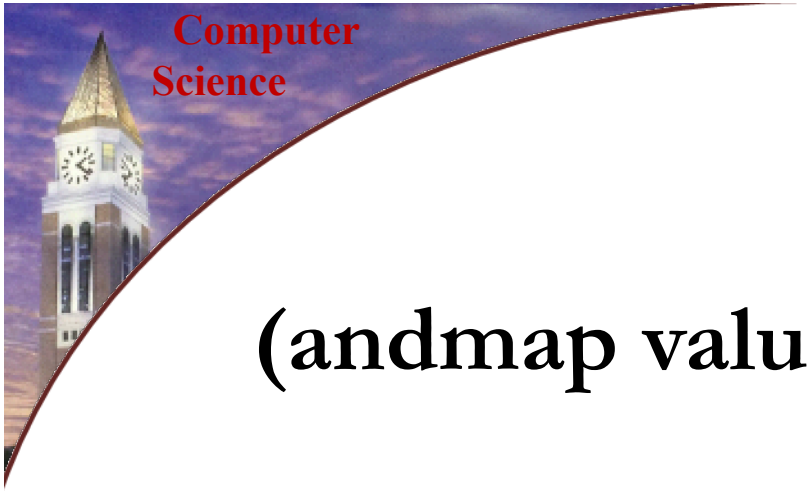
For a program consisting of multiple expressions, the **last** expression's value will be the value of the overall program!



`map` is not enough!

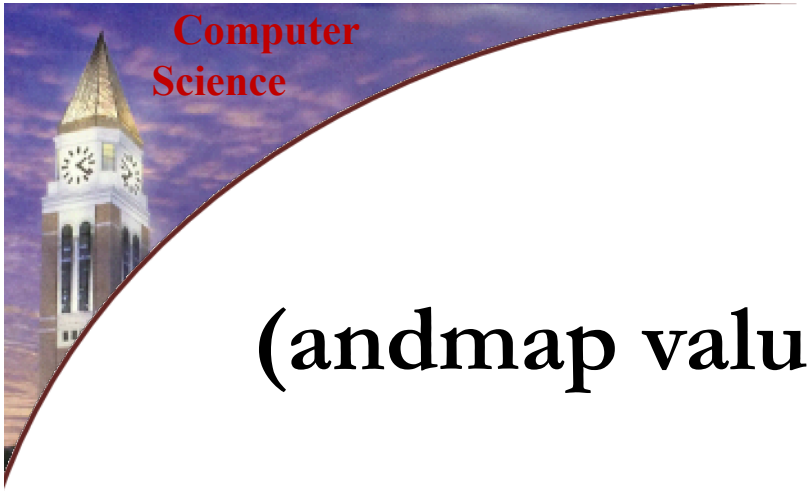


andmap !



(andmap value-of-expr list-of-expr)

andmap will apply **value-of-expr** to all the expressions in **list-of-expr**, and return the value of **the last expression** in the list ! Which is exactly what we need to evaluate a program which may consists of multiple expressions!

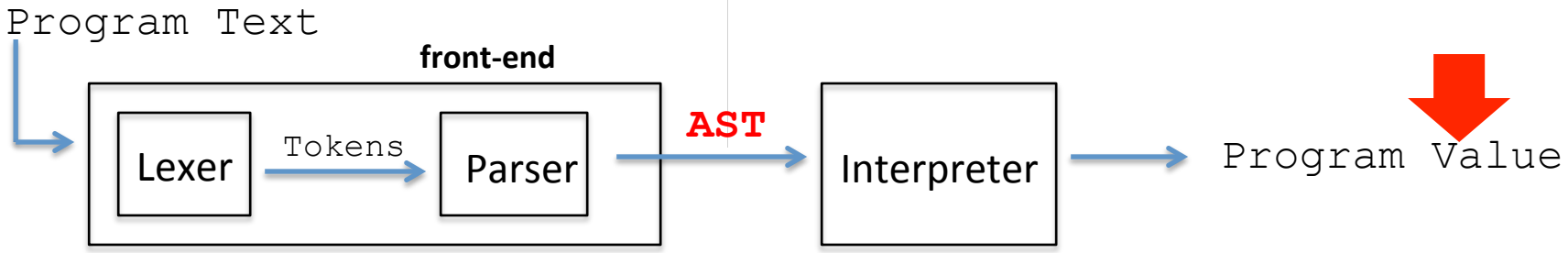


using `flatlist` function provided in hw06

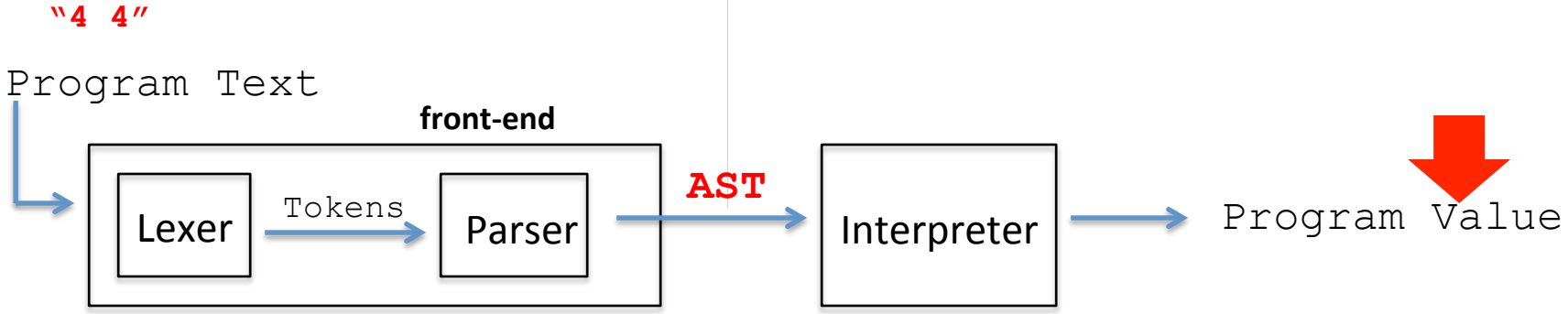
(andmap value-of-expr list-of-expr)

andmap will apply **value-of-expr** to all the expressions in **list-of-expr**, and return the value of **the last expression** in the list ! Which is exactly what we need to evaluate a program which may consists of multiple expressions!

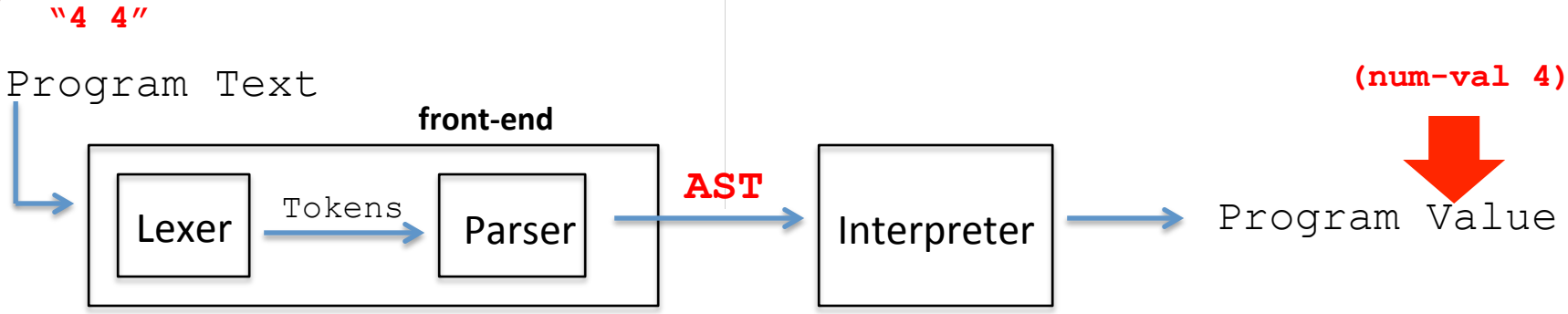
Internal Representation of Program Values



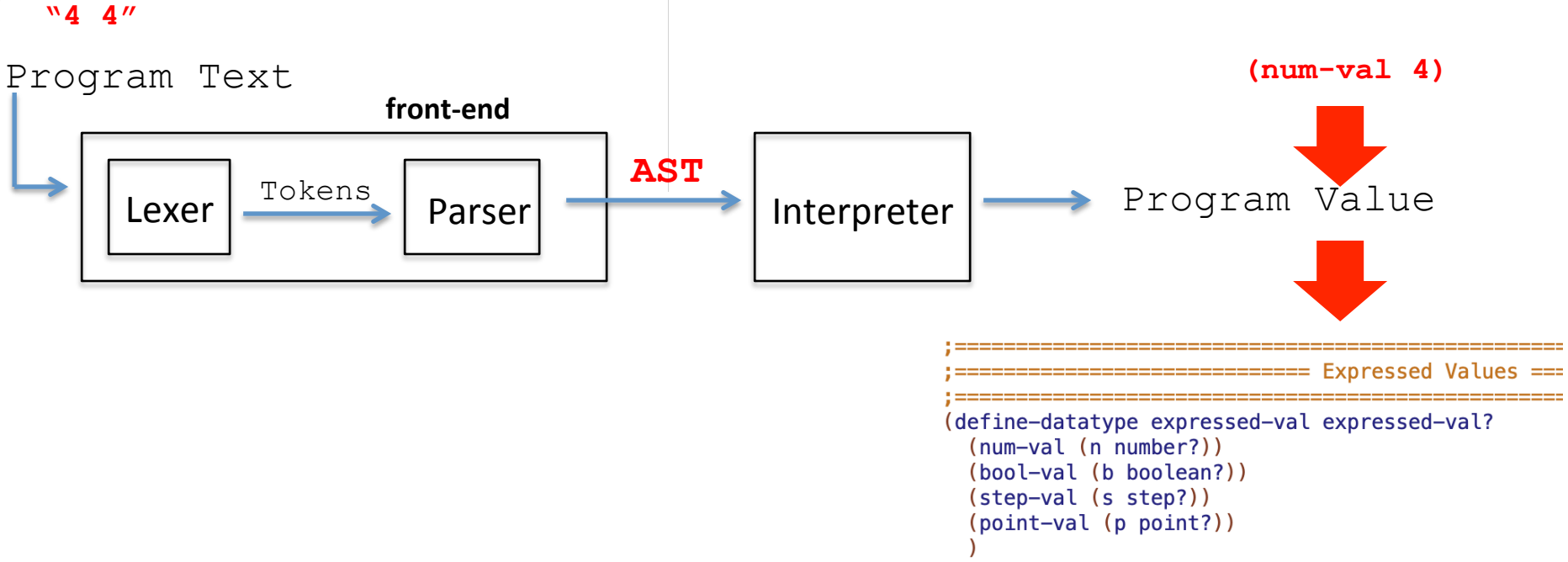
Internal Representation of Program Values



Internal Representation of Program Values

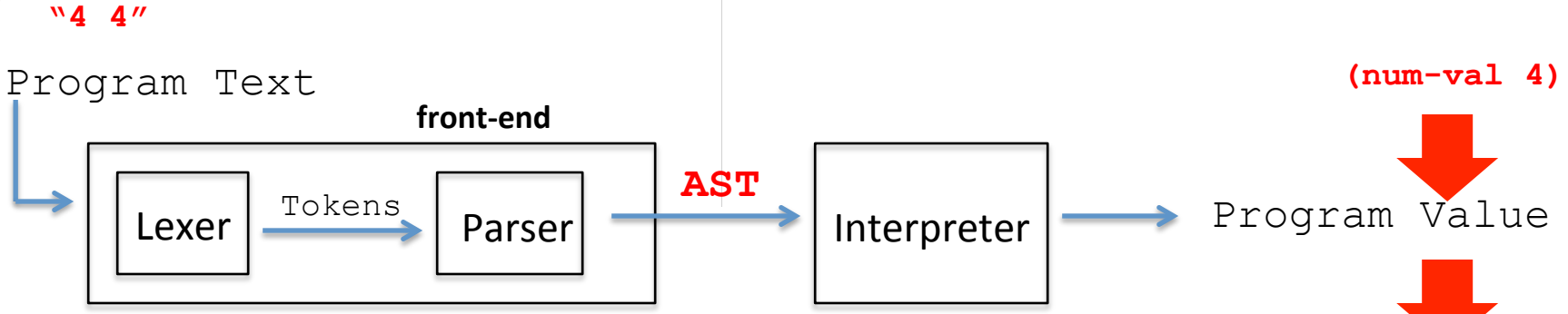


Internal Representation of Program Values




```
;;================================================
;;============================== Expressed Values ===
;;================================================
(define-datatype expressed-val expressed-val?
  (num-val (n number?))
  (bool-val (b boolean?))
  (step-val (s step?))
  (point-val (p point?))
  )
```

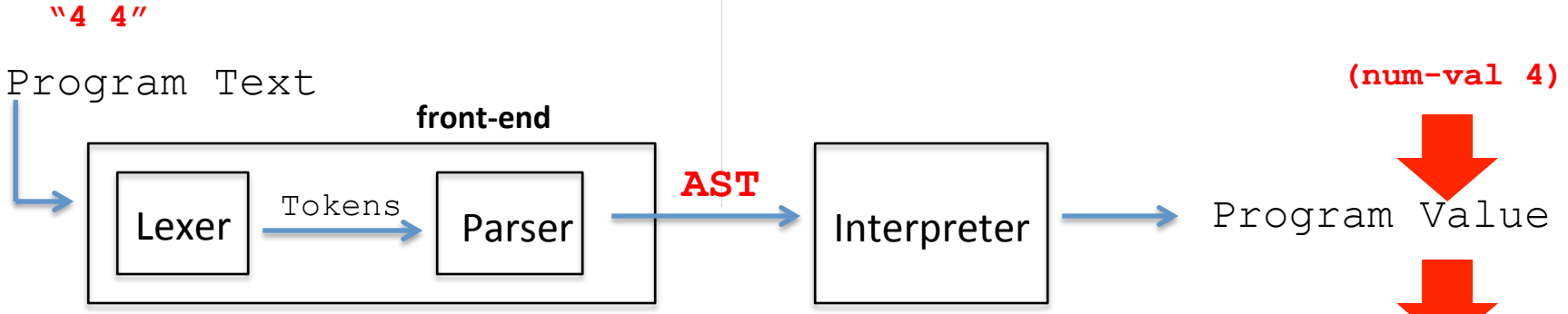
Internal Representation of Program Values




```
;;================================================
;;============================== Expressed Values ===
;;================================================
(define-datatype expressed-val expressed-val?
  (num-val (n number?))
  (bool-val (b boolean?))
  (step-val (s step?))
  (point-val (p point?))
  )
```

see ***hw06-env-values.rkt***

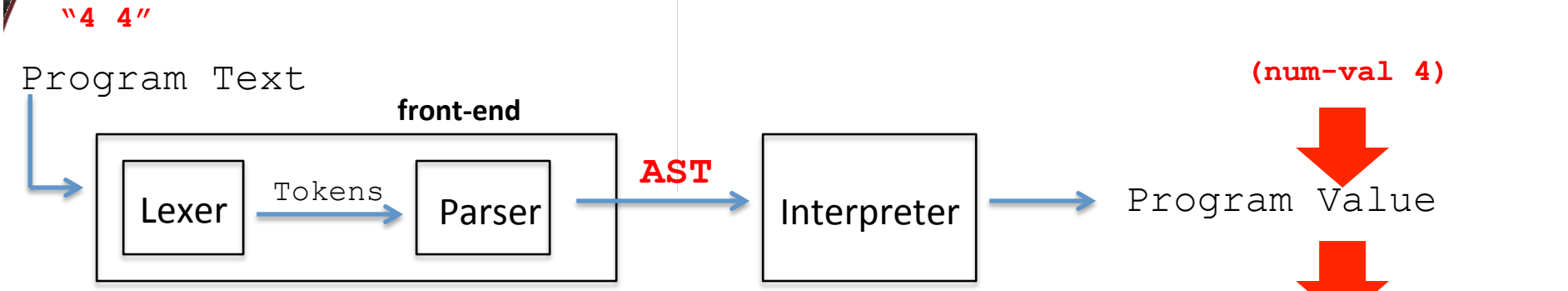
Internal Representation of Program Values




```
;;================================================
;;============================ Expressed Values ===
;;================================================
(define-datatype expressed-val expressed-val?
  (num-val (n number?))
  (bool-val (b boolean?))
  (step-val (s step?))
  (point-val (p point?))
  )
```

see ***hw06-env-values.rkt***

Internal Representation of Program Values

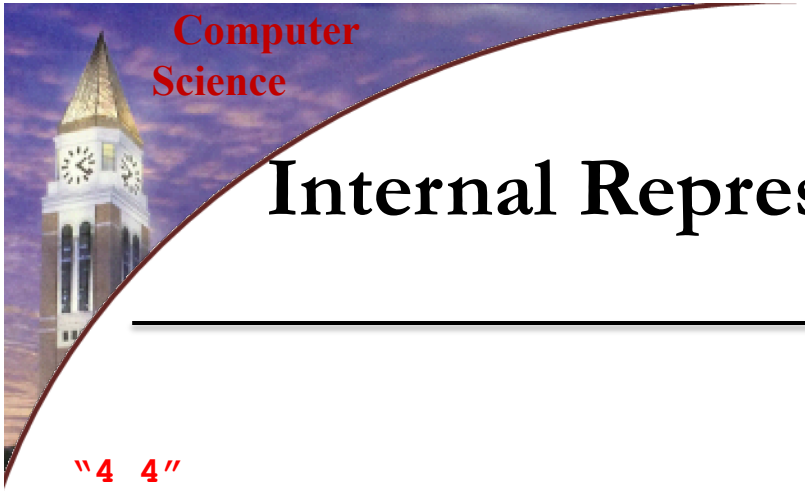



```
(define (num-val? num)
  (and (expressed-val? num)
       (cases expressed-val num
         (num-val (n) 3)
         (else #f))
       )
  )
```

```
(define (step-val->st st)
  (or (expressed-val? st) (invalid-args-exception "step-val->st" "expressed-val?" st))
  (cases expressed-val st
    (step-val (st) st)
    (else (invalid-args-exception "step-val->st" "num-val?" st))
    )
  )
```

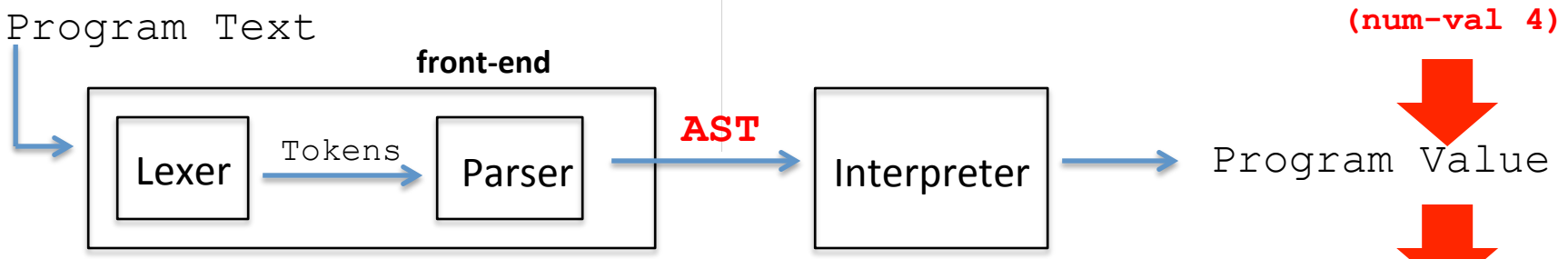
```
;;================================================
;;============================ Expressed Values ===
;;================================================
(define-datatype expressed-val expressed-val?
  (num-val (n number?))
  (bool-val (b boolean?))
  (step-val (s step?))
  (point-val (p point?))
  )
```

see ***hw06-env-values.rkt***



Internal Representation of Program Values

"4 4"




```
(define (num-val? num)
  (and (expressed-val? num)
       (cases expressed-val num
         (num-val (n) 3)
         (else #f))
       )
  )
```

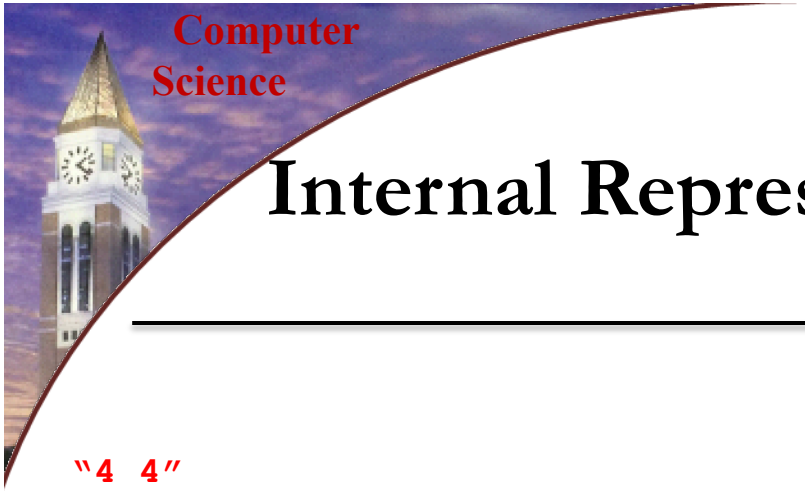
predicate for `num-val`

```
(define (step-val->st st)
  (or (expressed-val? st) (invalid-args-exception "step-val->st" "expressed-val?" st))
  (cases expressed-val st
    (step-val (st) st)
    (else (invalid-args-exception "step-val->st" "num-val?" st))
    )
  )
```

extractor for `num-val`

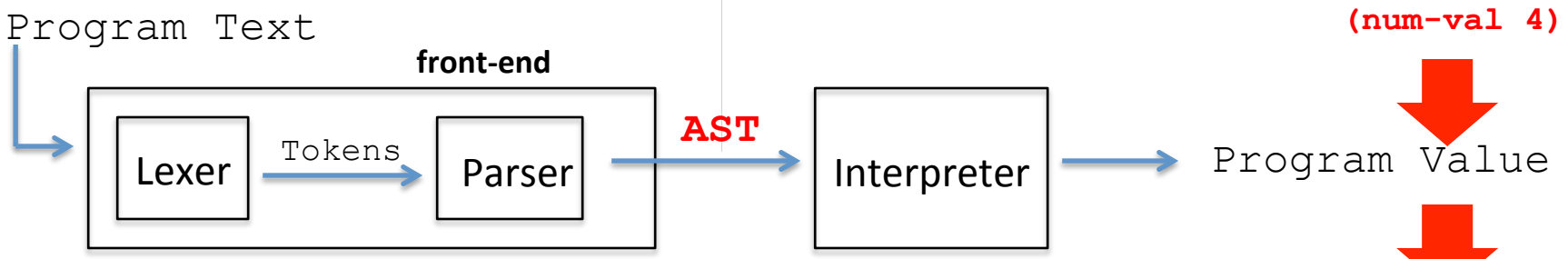
```
;;================================================
;;============================ Expressed Values ===
;;================================================
(define-datatype expressed-val expressed-val?
  (num-val (n number?))
  (bool-val (b boolean?))
  (step-val (s step?))
  (point-val (p point?))
  )
```

see ***hw06-env-values.rkt***



Internal Representation of Program Values

"4 4"




```
(define (num-val? num)
  (and (expressed-val? num)
       (cases expressed-val num
         (num-val (n) 3)
         (else #f))
       )
  )
```

predicate for `num-val`

similar to Integer boxing in Java

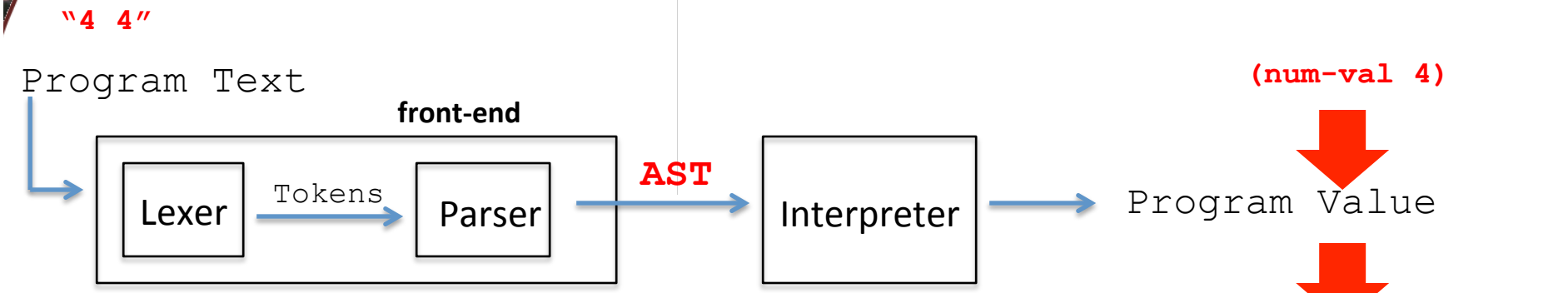
```
(define (step-val->st st)
  (or (expressed-val? st) (invalid-args-exception "step-val->st" "expressed-val?" st))
  (cases expressed-val st
    (step-val (st) st)
    (else (invalid-args-exception "step-val->st" "num-val?" st))
    )
  )
```

extractor for `num-val`

```
;;================================================
;;============================ Expressed Values ===
;;================================================
(define-datatype expressed-val expressed-val?
  (num-val (n number?))
  (bool-val (b boolean?))
  (step-val (s step?))
  (point-val (p point?))
  )
```

see ***hw06-env-values.rkt***

Internal Representation of Program Values




```
(define (num-val? num)
  (and (expressed-val? num)
       (cases expressed-val num
         (num-val (n) 3)
         (else #f))
       )
  )
```

predicate for `num-val`

similar to Integer boxing in Java

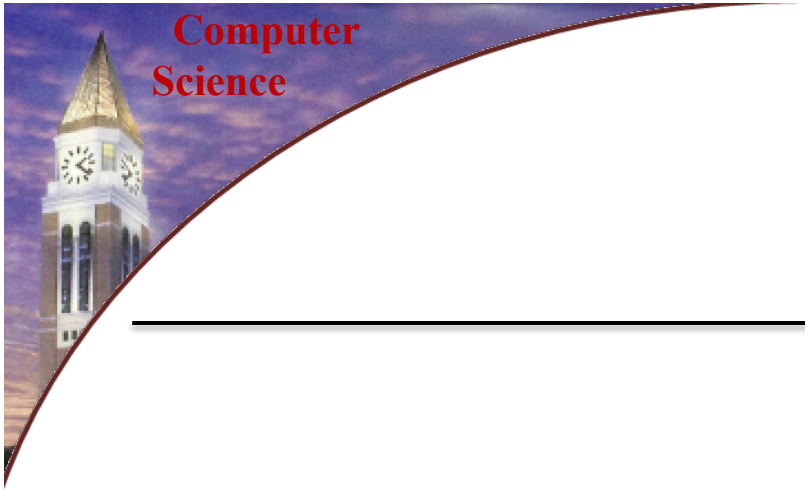
```
(define (step-val->st st)
  (or (expressed-val? st) (invalid-args-exception "step-val->st" "expressed-val?" st))
  (cases expressed-val st
    (step-val (st) st)
    (else (invalid-args-exception "step-val->st" "num-val?" st))
    )
  )
```

extractor for `num-val`

similar to Integer unboxing in Java

```
;;================================================
;;============================ Expressed Values ===
;;================================================
(define-datatype expressed-val expressed-val?
  (num-val (n number?))
  (bool-val (b boolean?))
  (step-val (s step?))
  (point-val (p point?))
  )
```

see ***hw06-env-values.rkt***



HW06

parser = (parser-generator lexical-spec grammar-spec)

ast = (parser a-program-string)

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"
            |(<expr> <expr>)      "point-expr"
```

Original Grammar in BNF

```
(define the-grammar
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-expr)
```

BNF Grammar's Scheme Implementation

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"
```

The name of each production rule

```
(define the-grammar
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-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"
            |(<expr> <expr>)        "point-expr"
```

The name of each production rule

```
(define the-grammar
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-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"
            |(<expr> <expr>)        "point-expr"
```

The name of each production rule

```
(define the-grammar
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-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"
            |(<expr> <expr>)        "point-expr"
```

The name of each production rule

```
(define the-grammar
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-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"
            |(<expr> <expr>)        "point-expr"
```

The name of each production rule

```
(define the-grammar
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-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"
            |(<expr> <expr>)        "point-expr"
```

The name of each production rule

```
(define the-grammar
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-expr)
```

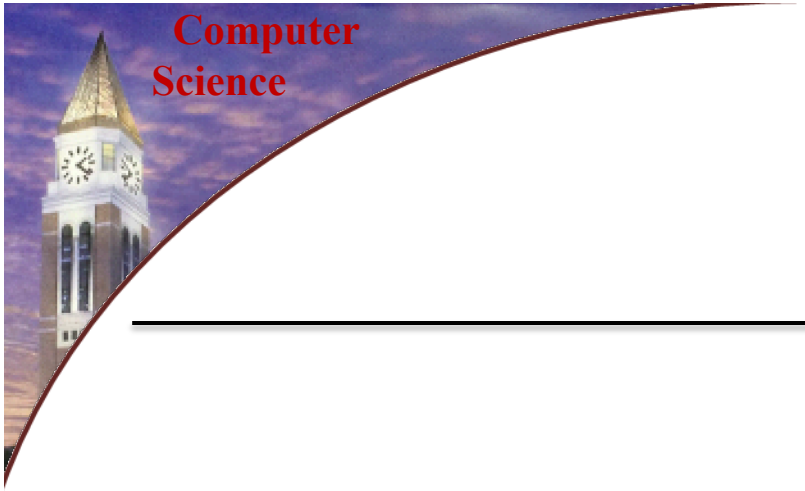
HW06

(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"
            |(<expr> <expr>)        "point-expr"
```

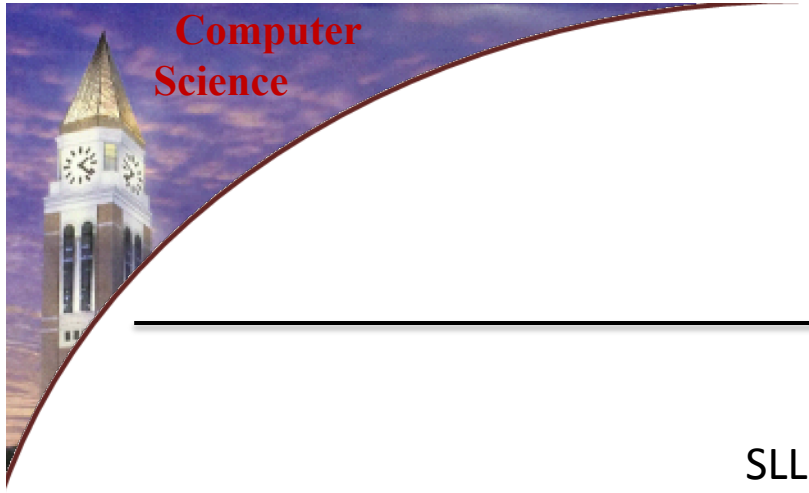
The name of each production rule

```
(define the-grammar
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-expr)
```




```
parser = (parser-generator lexical-spec grammar-spec)
```

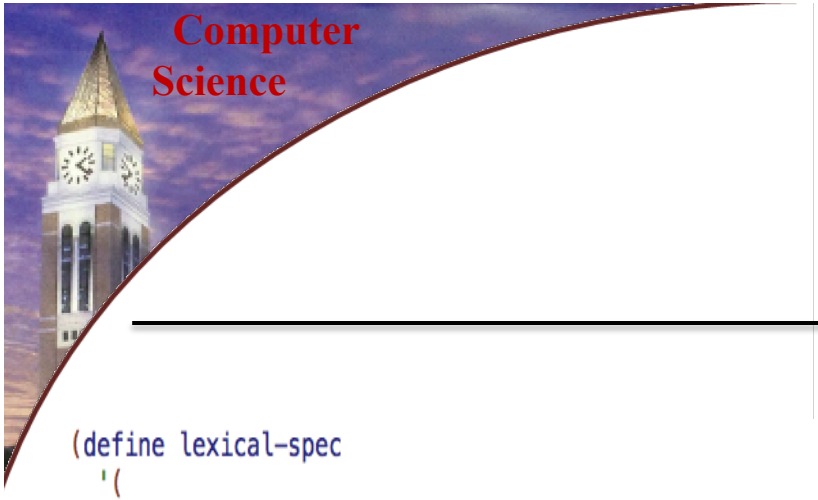
```
ast = (parser a-program-string)
```



SLLGEN Boiler Plate Code (**Slide 18**)

parser = (sllgen:make-string-parser `lexical-spec grammar-spec`)

ast = (parser a-program-string)



HW06

(Problem 3)

```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comment ("#" (arbno (not #\newline))) skip)
    (number (digit (arbno digit)) number)
    (number ("-" digit (arbno digit)) number)

    (identifier (letter (arbno (or letter digit "_" "-" "?"))) symbol)
    )
  )
```

```
(define grammar-spec
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-expr)))
```

```
(define parser
   (sllgen:make-string-parser lexical-spec grammar-spec )
  )
```

HW06

```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comment ("#" (arbno (not #\newline))) skip)
    (number (digit (arbno digit)) number)
    (number ("-" digit (arbno digit)) number)

    (identifier (letter (arbno (or letter digit "_" "-" "?"))) symbol)
    )
  )
```

```
(define grammar-spec
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-expr)))
```

```
(define parser
   (sllgen:make-string-parser lexical-spec grammar-spec )
)
```

sllgen parser generator

HW06

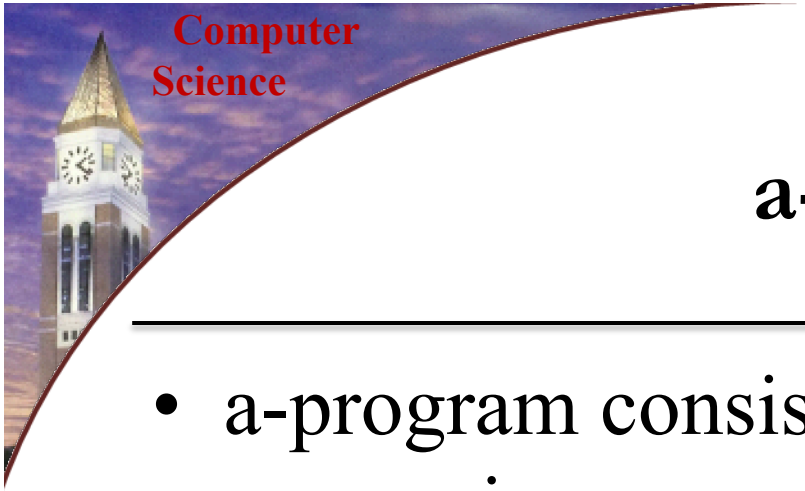
```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comment ("#" (arbno (not #\newline))) skip)
    (number (digit (arbno digit)) number)
    (number ("-" digit (arbno digit)) number)

    (identifier (letter (arbno (or letter digit "_" "-" "?"))) symbol)
    )
  )
```

```
(define grammar-spec
  '(
    (program (expr (arbno expr)) a-program)

    (expr (number) num-expr)
    (expr ("up" "(" expr ")") up-expr)
    (expr ("down" "(" expr ")") down-expr)
    (expr ("left" "(" expr ")") left-expr)
    (expr ("right" "(" expr ")") right-expr)
    (expr ("(" expr expr ")") point-expr)))
```

```
(define parser
   (sllgen:make-string-parser lexical-spec grammar-spec )
)
```

sllgen parser generator

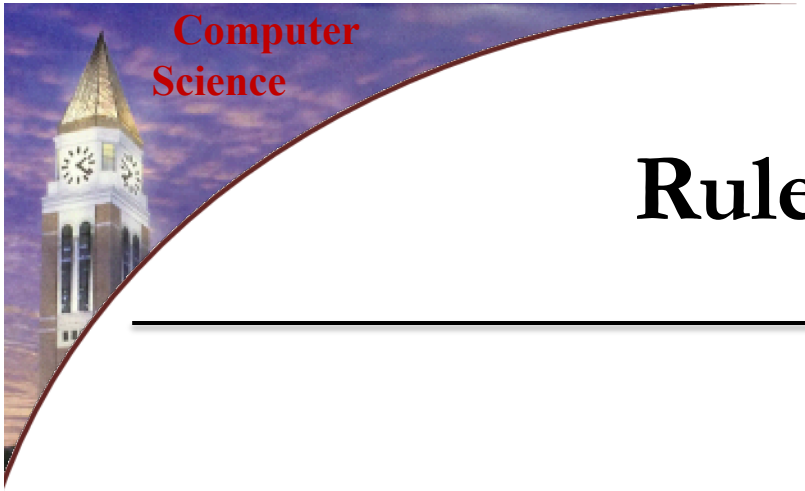
`parser` is automatically generated by `sllgen`!



a-program

- a-program consists of one or more expressions

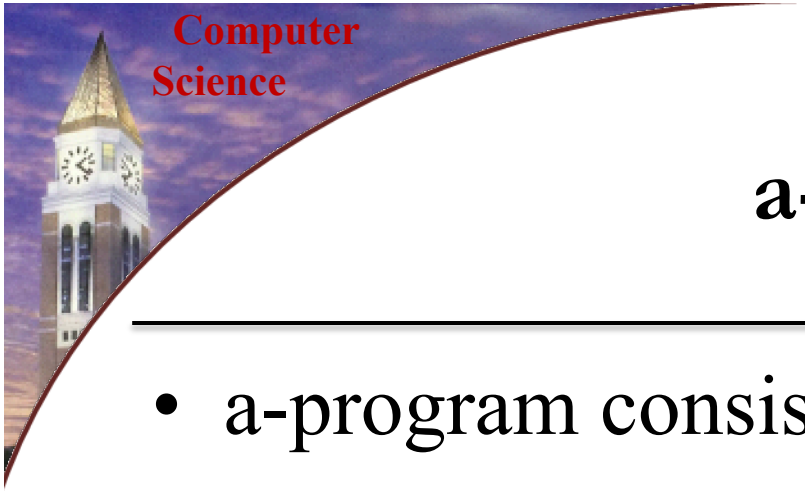
Let's first think of the "**more expressions**" case!



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

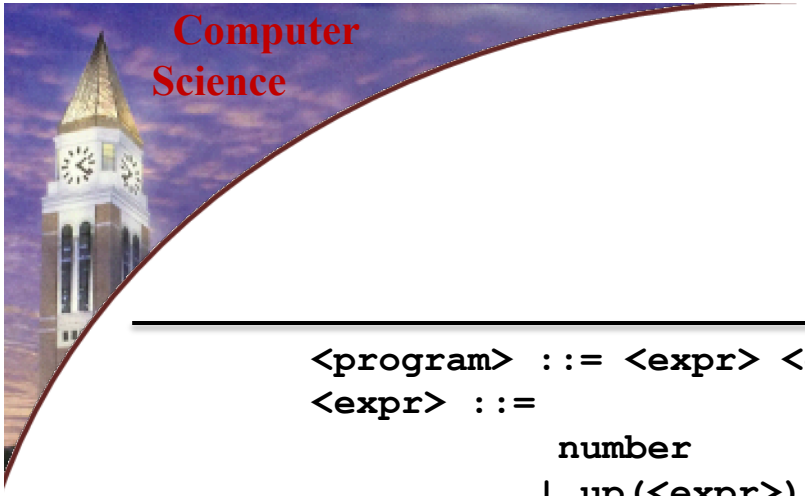
```
(run "42")
```

```
(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"
```

HW06

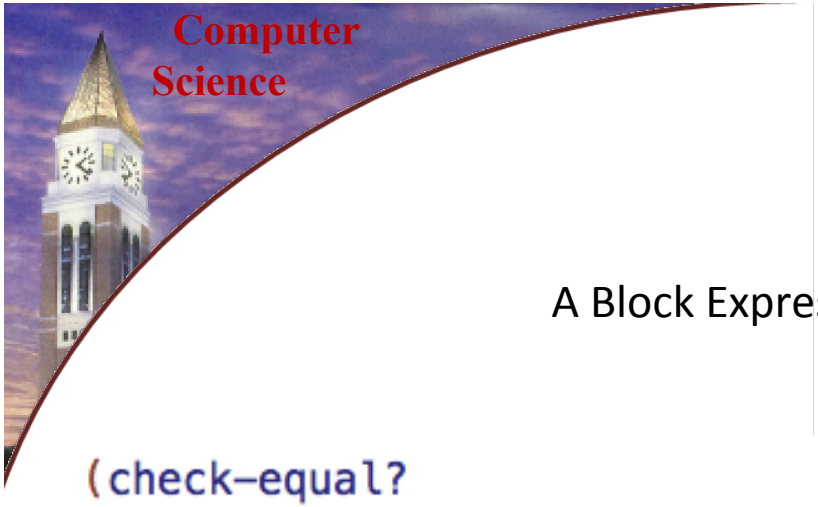
(Problem 4)

```
<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"
```

HW06

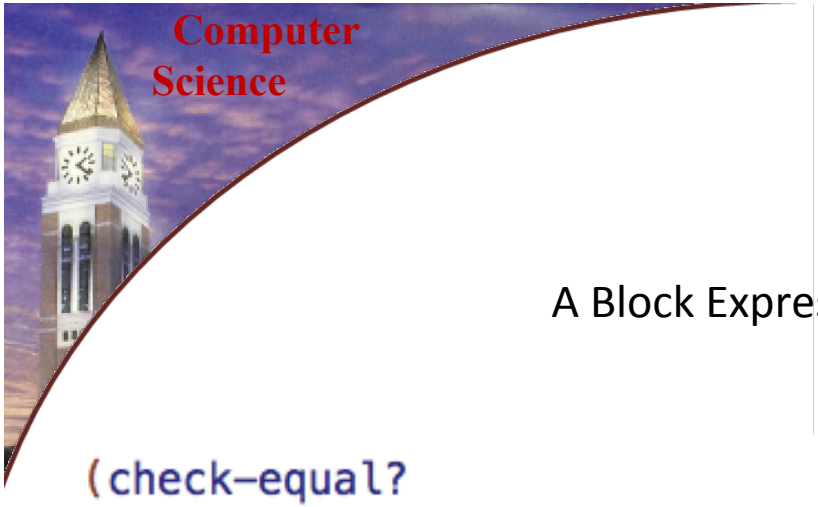
(Problem 4)

```
<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"
 )
```

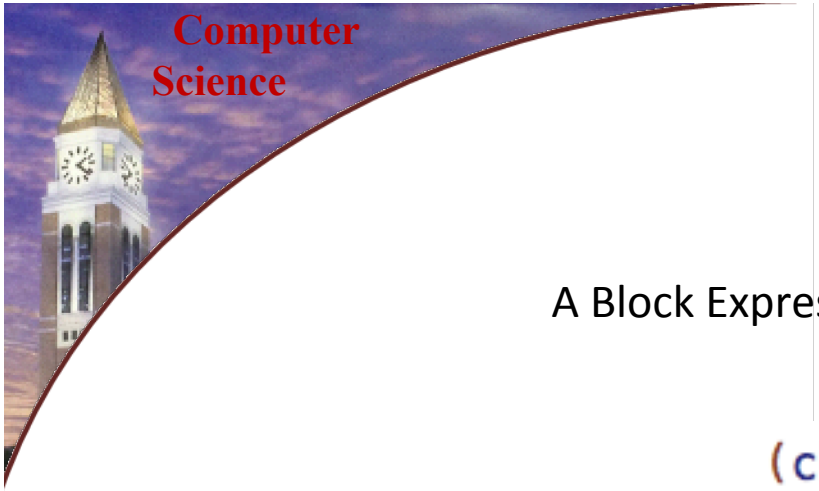


A Block Expression Example (no nested blocks)

```
(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`

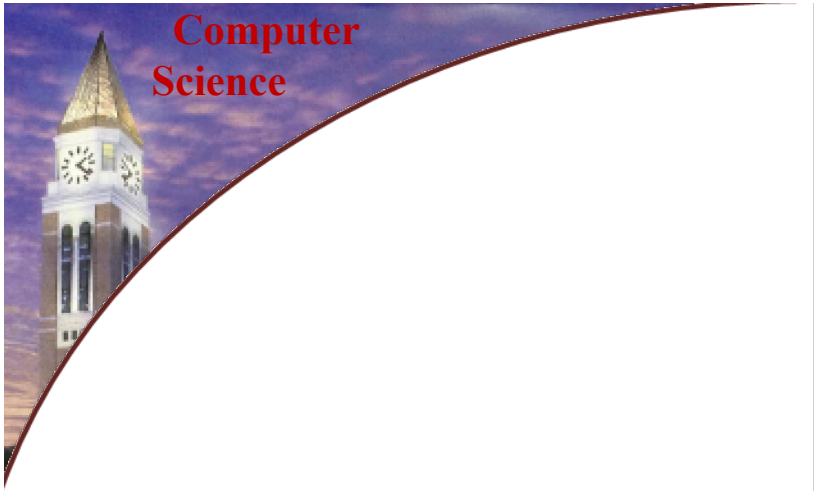


A Block Expression Example (with nested blocks)

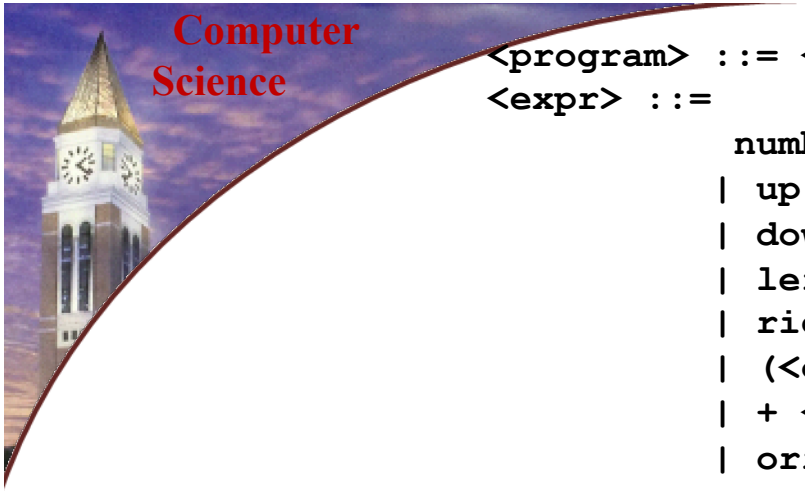
```
(check-equal?
 (run
  "{
     val x = 42
     {
      val y = 23
      x
     }
  }")
 (num-val 42)
 )
```

Nested Block Expression

Outer Block Expression

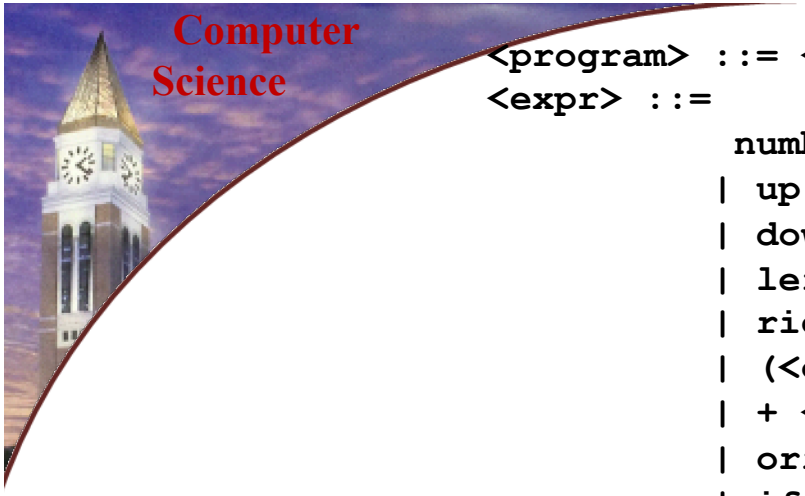



```
(run "{
        val x = 42
        x
        val y = 33


      }"))
```




```
<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"
```

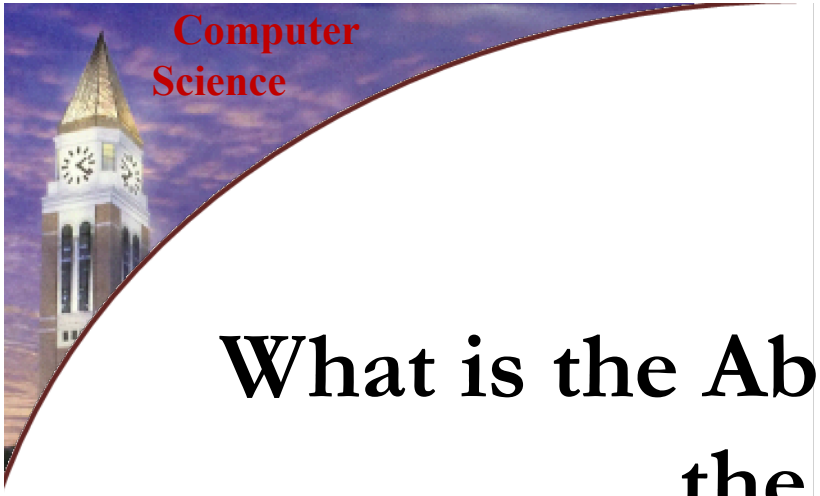
```
(run "{
        val x = 42
        x
        val y = 33


      }"))
```




```
<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"
```

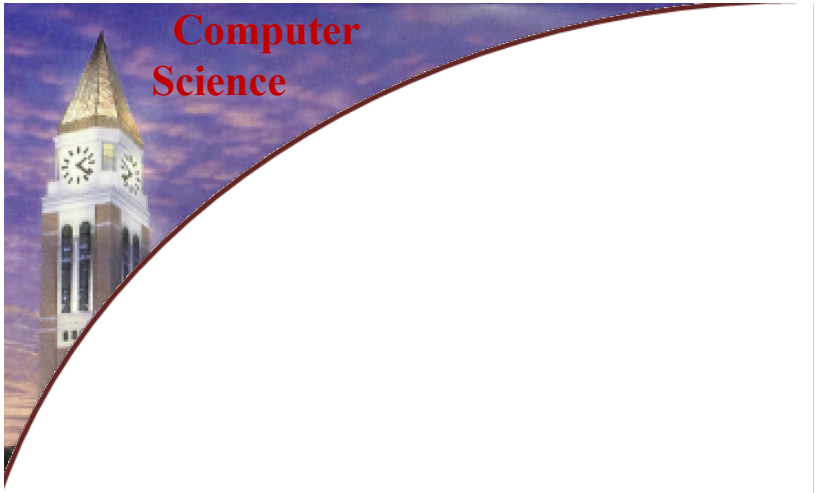
```
(run "{
        val x = 42
        x
        val y = 33
      }"))
```

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



What is the Abstract Syntax Tree For the following?

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



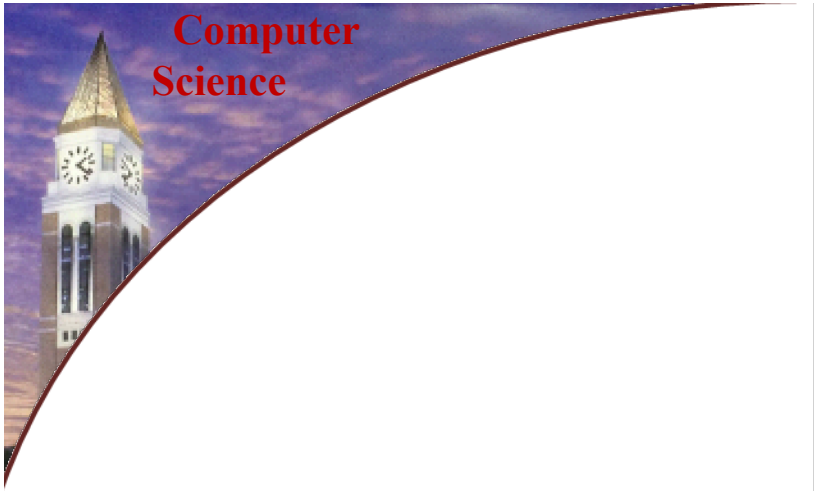

```
"{
  val x = 42
  {
   val x = 23
  }
  x
 }"
```

val-expr

expr (block-expr is a variant of expr)

expr

expr (block-expr is a variant of expr)



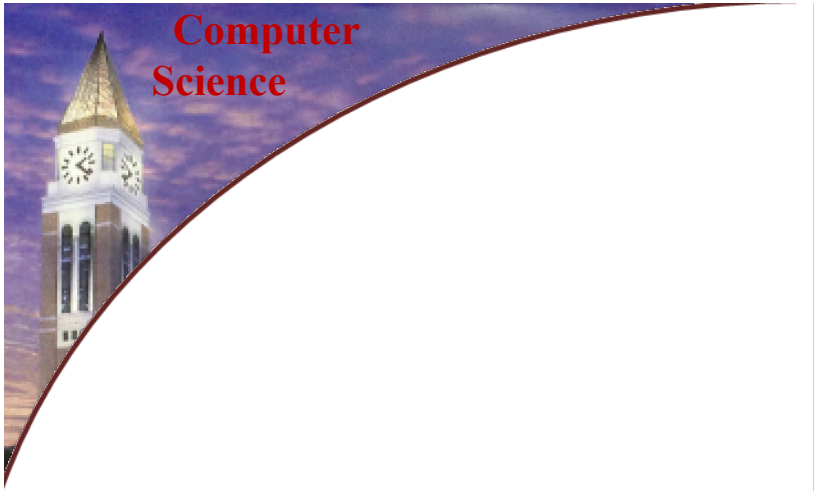
HW06

(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 !

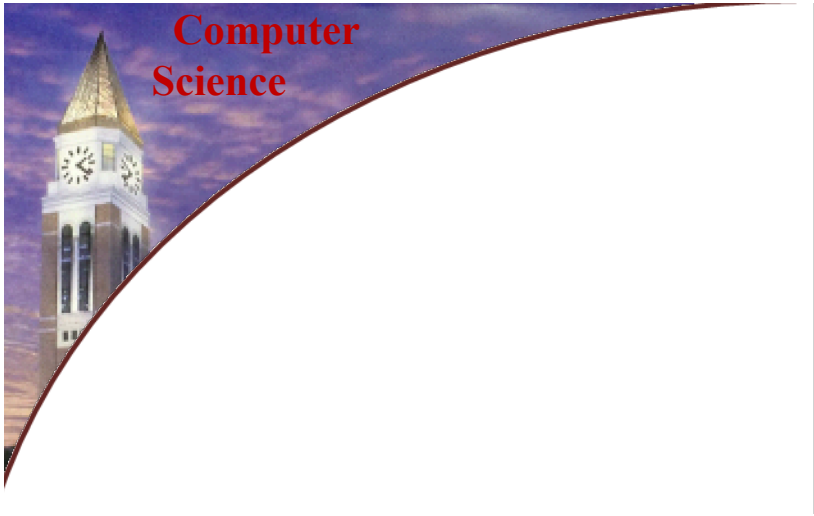


HW06

(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)))
      )
  )
```

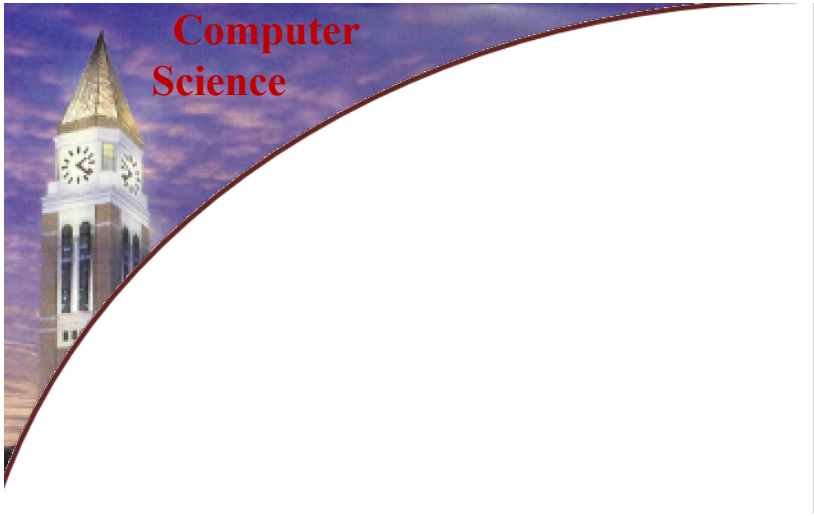


HW06

(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)))
      )
  )
```

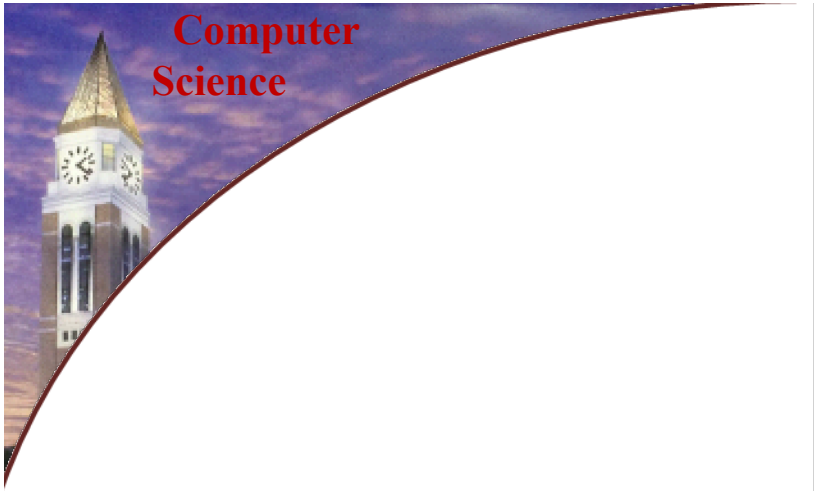


HW06

(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)




  )
```

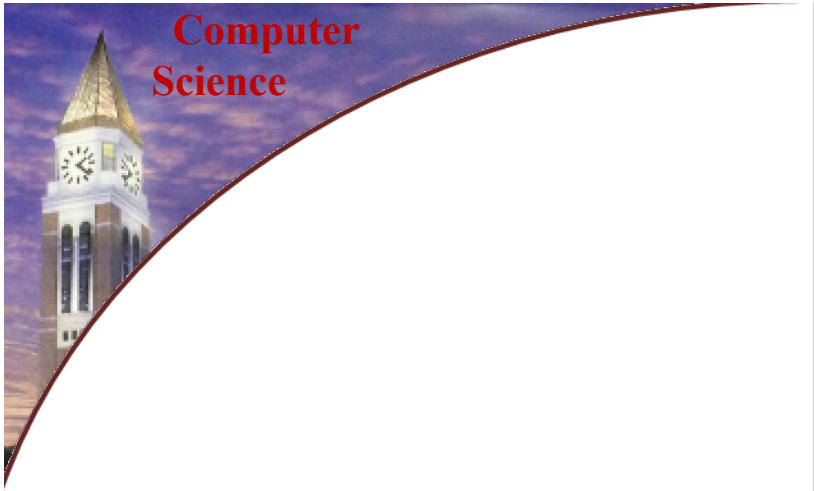


HW06

(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




  )
```

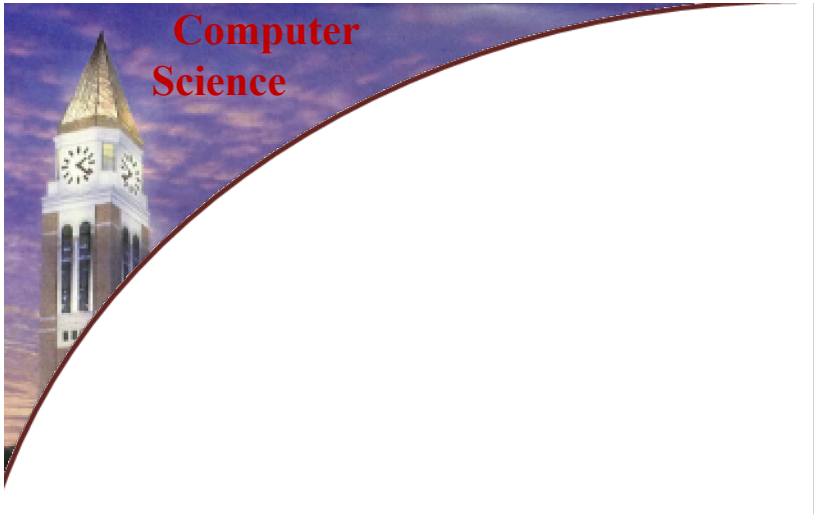


HW06

(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)




  )
```

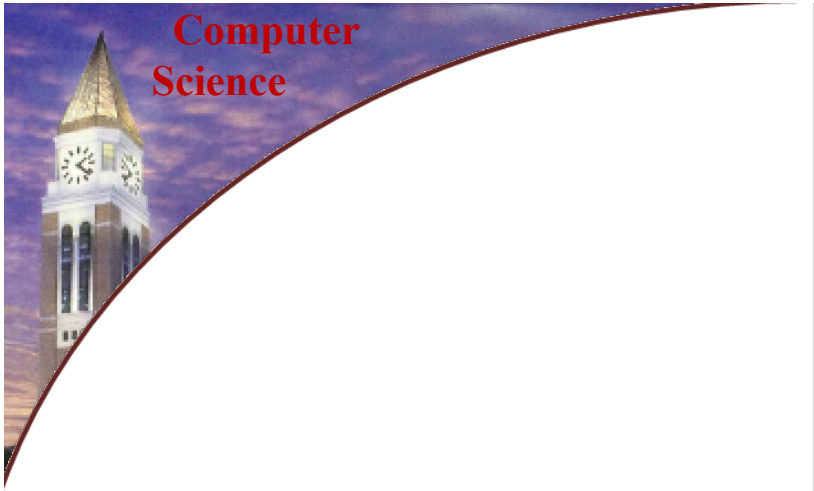


HW06

(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)]



  )
```

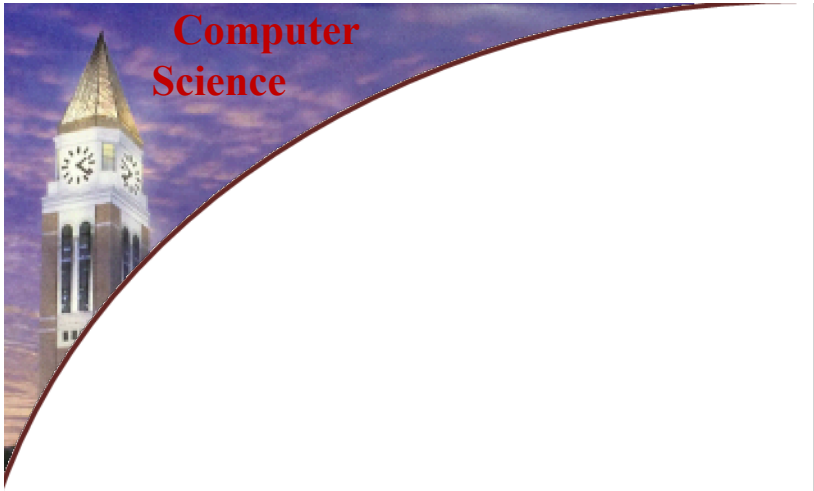


HW06

(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)



  )
```

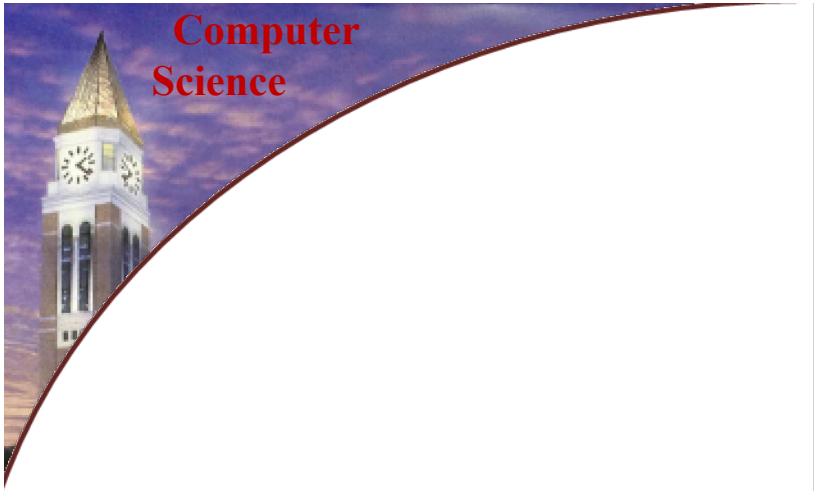


HW06

(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)]



  )
```

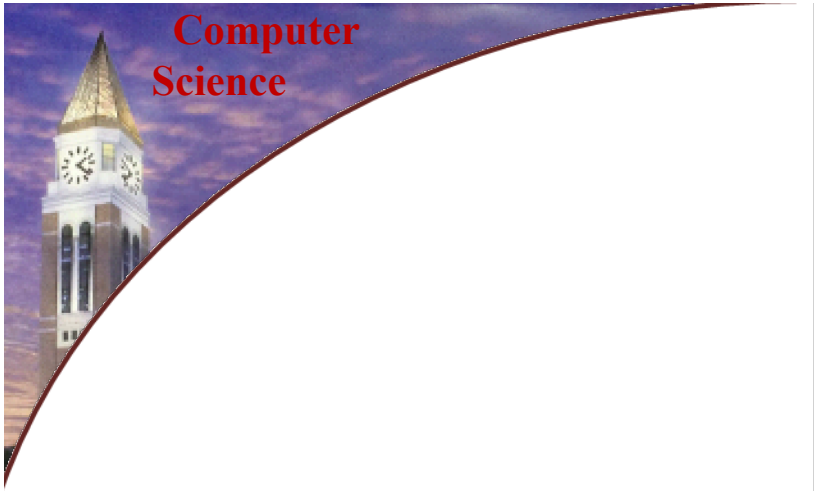


HW06

(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)


  )
```

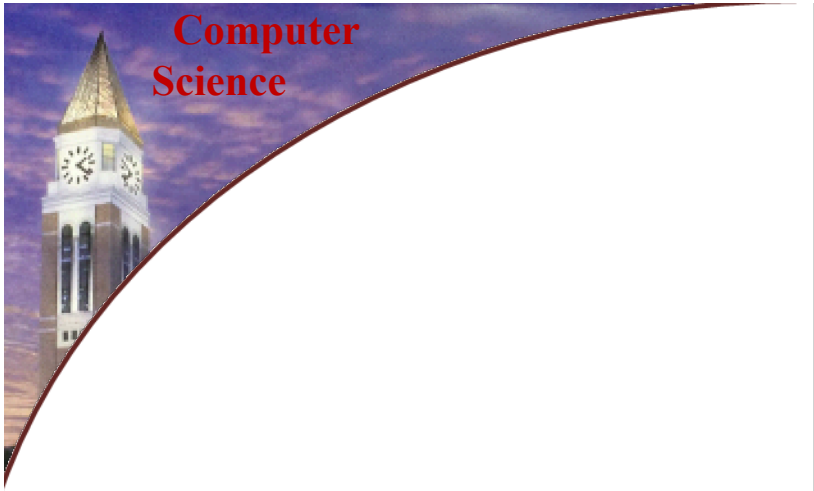


HW06

(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)]


  )
```



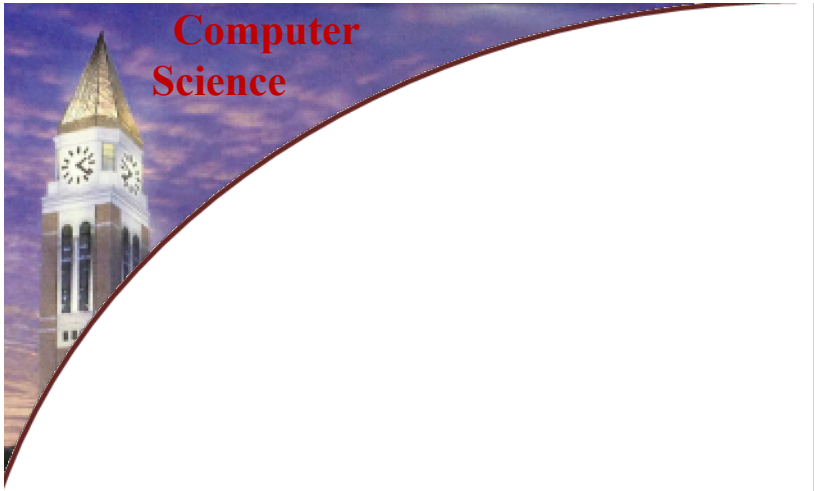
HW06

(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))   ; to kick off the interpreter here !
      (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))
     )
    )
  )
```