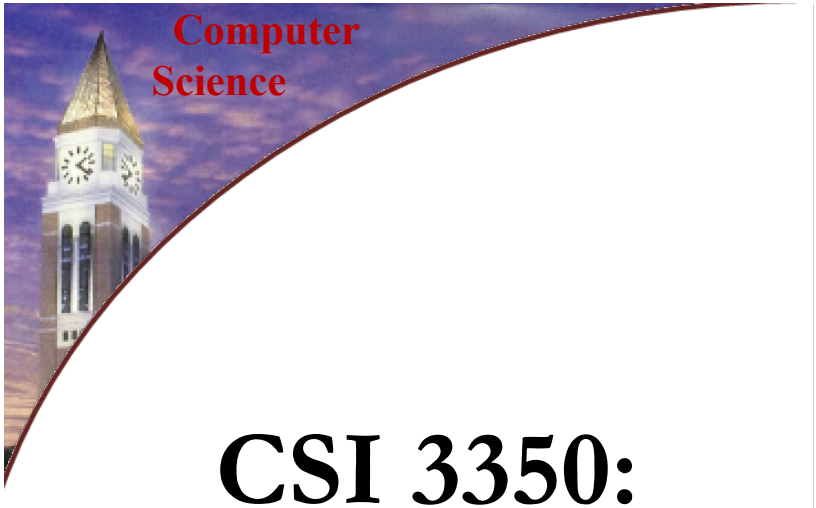
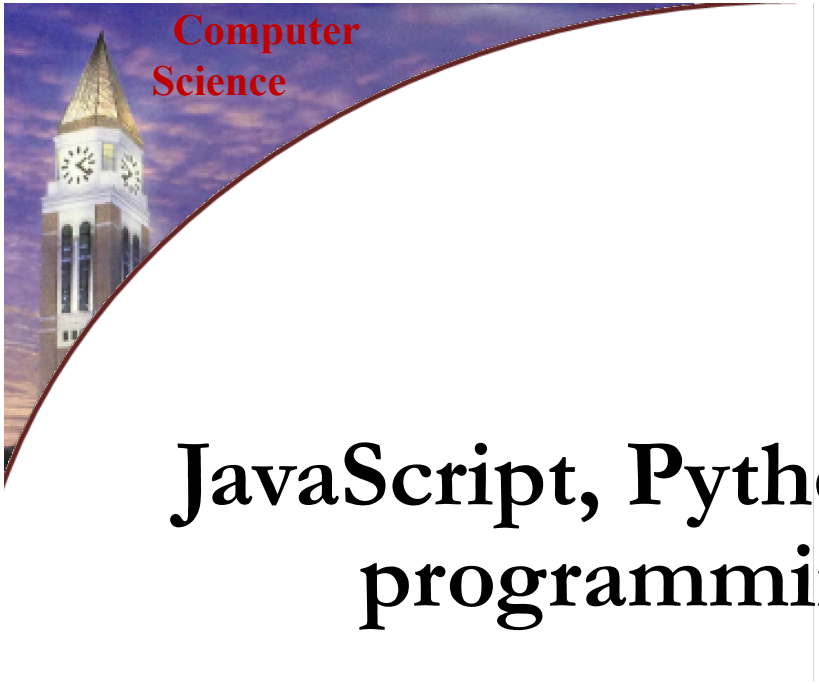


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

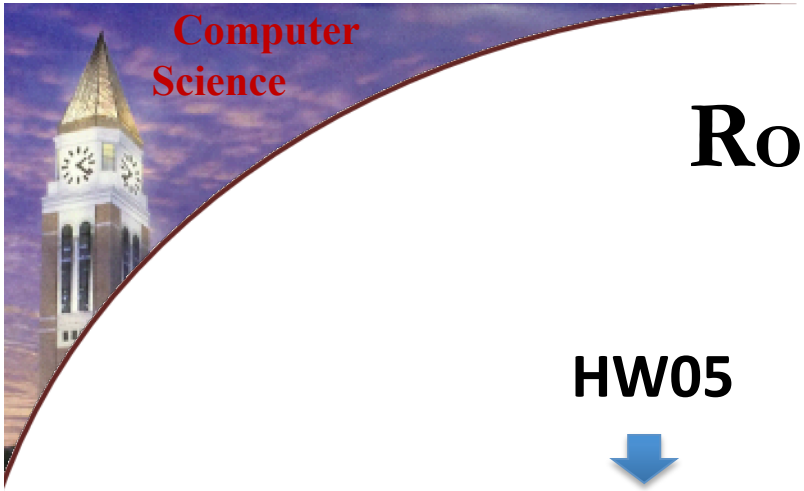
Oakland University



JavaScript, Python are the Most popular programming languages now !

JavaScript, Python are the Most popular programming language now !

functional programming is greatly supported in JavaScript, Python !



Road Ahead -

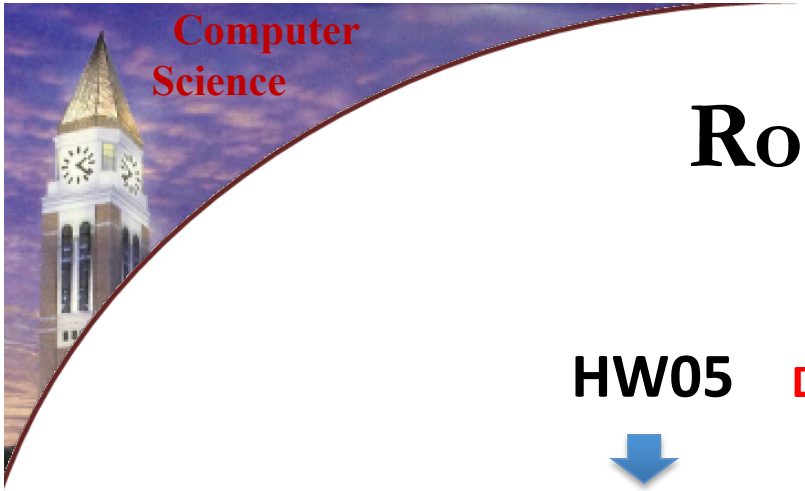
HW05

HW06

Exam02

HW07

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



Road Ahead -

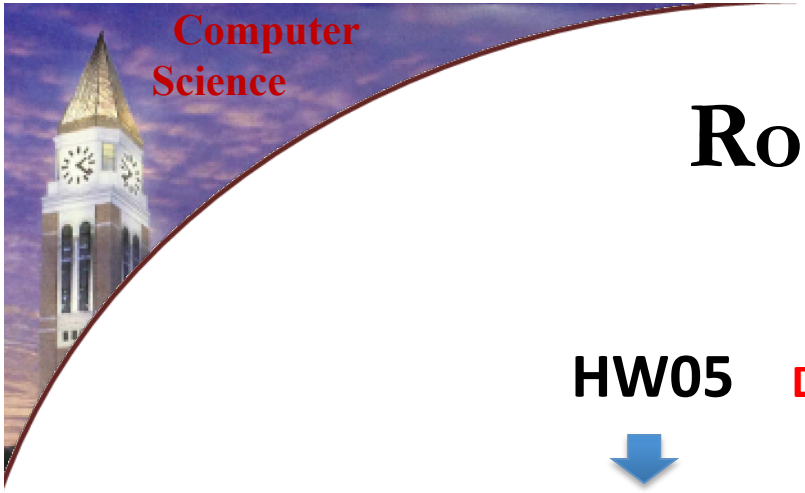
HW05 **Due: Nov 12**

HW06

Exam02

HW07

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



Road Ahead -

HW05 Due: Nov 12

↓

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

↓

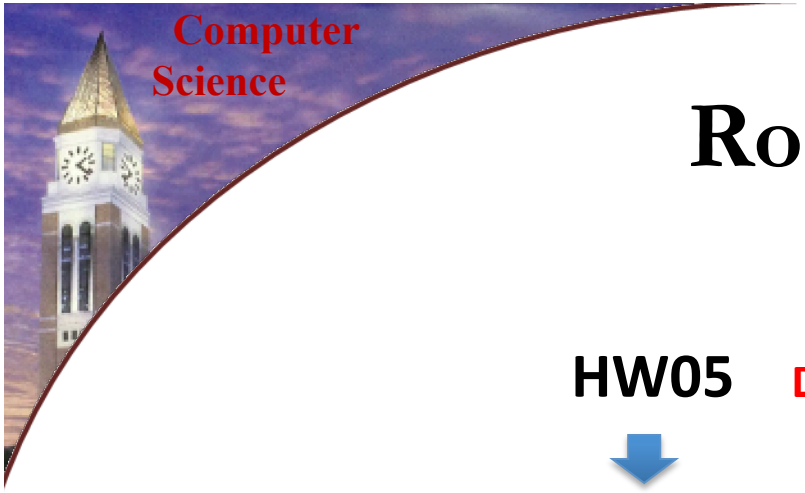
Exam02

↓

HW07

↓

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



Road Ahead -

HW05 Due: Nov 12

↓

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

↓

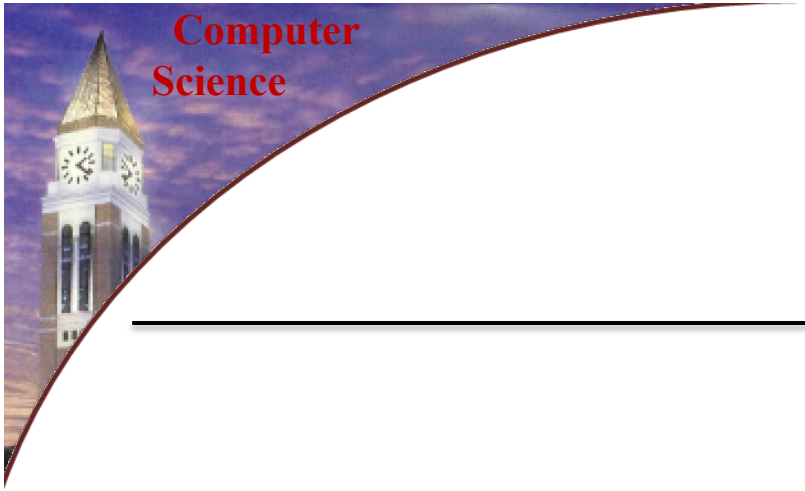
Exam02 ← **Nov 27**

↓

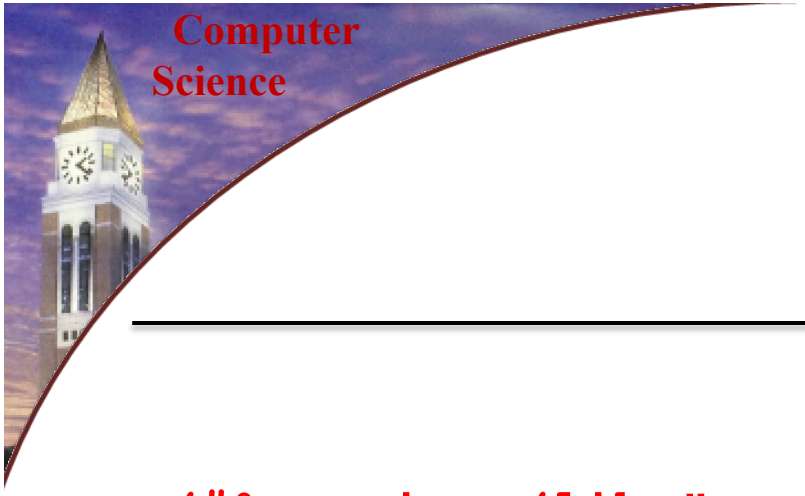
HW07

↓

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



A Tool for Defining Data Types (less grunt work!)




```
(#%require (lib "eopl.ss" "eopl"))
```

define-datatype

define-datatype

```
Env ::= (empty-env )
        | (extend-env var val Env)

(define-datatype Env Env?
   (empty-env)
   (extend-env (var symbol?)(val number?)(env Env?))
)
```

name of the 2nd variant

name of the 1st field of the 2nd variant

name of the 2nd field of the 2nd variant

name of the 3rd field of the 2nd variant

define-datatype

```
Env ::= (empty-env )
        | (extend-env var val Env)
```

```
(define-datatype Env Env?
   (empty-env)
   (extend-env (var symbol?)(val number?)(Env Env?))
)
```

`var` needs to be of type `number`

`var` needs to be of type `symbol`

`Env` needs to be of type `Env`

define-datatype

```
Env ::= (empty-env )
        | (extend-env var val Env)
```

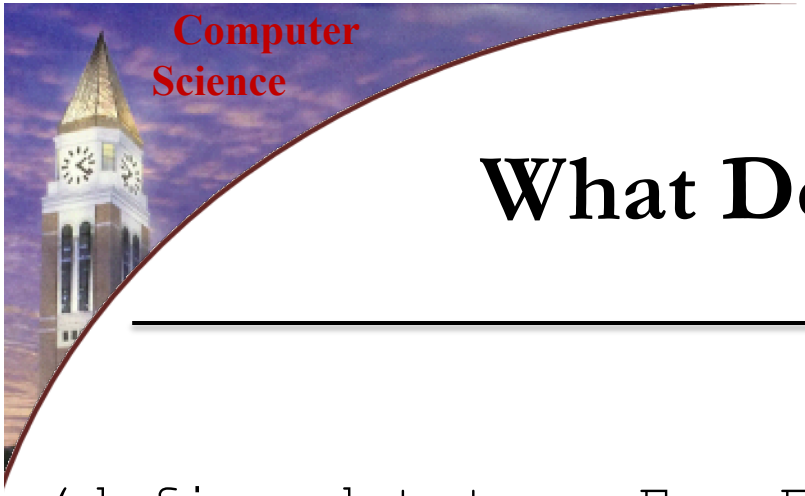
```
(define-datatype Env Env?
   (empty-env)
   (extend-env (var symbol?)(val number?)(Env Env?))
)
```

`var` needs to be of type `number`

`var` needs to be of type `symbol`

`Env` needs to be of type `Env`

(define-datatype type-name predicate-name
{ (variant-name { (field-name predicate)}*)}+



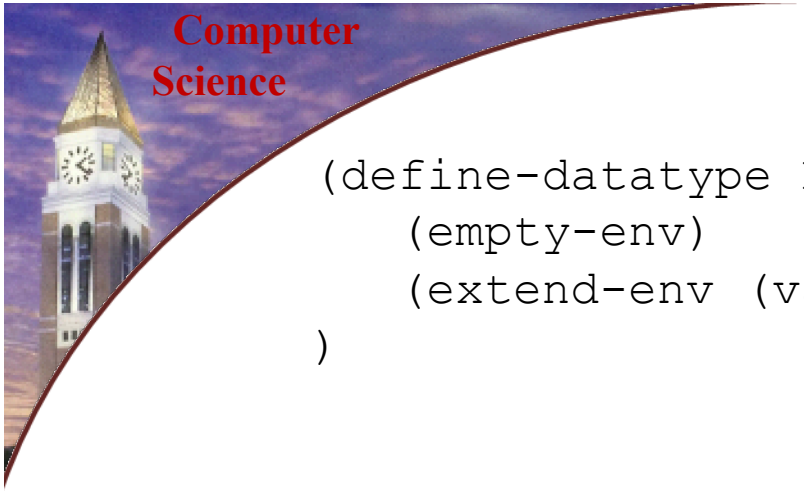
What Do We NOT Get?

```
(define-datatype Env Env?
    (empty-env)
    (extend-env (var symbol?)(val number?)(env Env?))
)
```

- We do NOT get
  - Extractors: `Env->var`, `Env->val`, `Env->env`
  - Predicates for variants: `empty-env?`, `extend-env?`

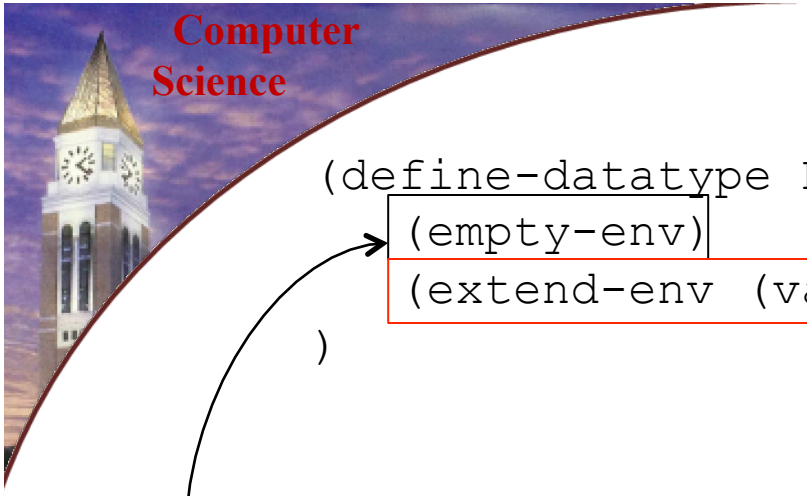
`cases` Syntax Abstraction

`cases` understands `define-datatype`



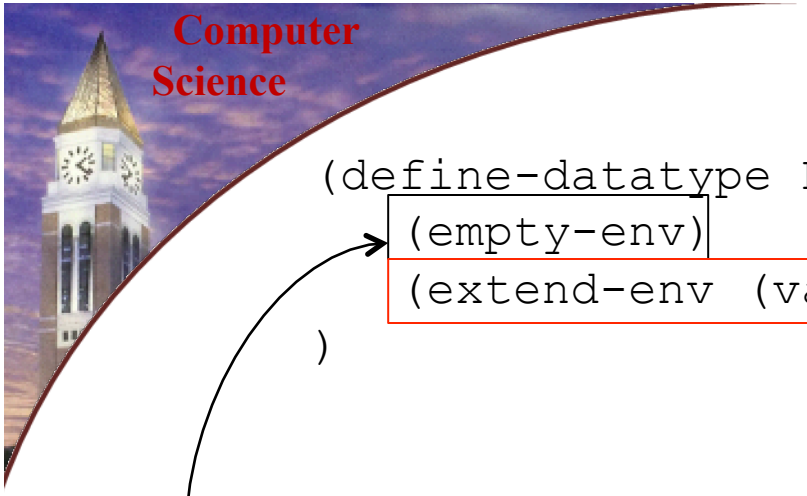

```
(define-datatype Env Env?
   (empty-env)
   (extend-env (var symbol?)(val number?)(env Env?))
)
```

```
(define (apply-env env search-var)
     (cases Env env
       (empty-env () (raise "No such variable found"))
       (extend-env
          (saved-var saved-val saved-env)
            (if (eqv? search-var saved-var)
                 saved-val
                (apply-env saved-env search-var)))))
```




```
(define-datatype Env Env?
   (empty-env)
   (extend-env (var symbol?)(val number?)(env Env?))
)
```

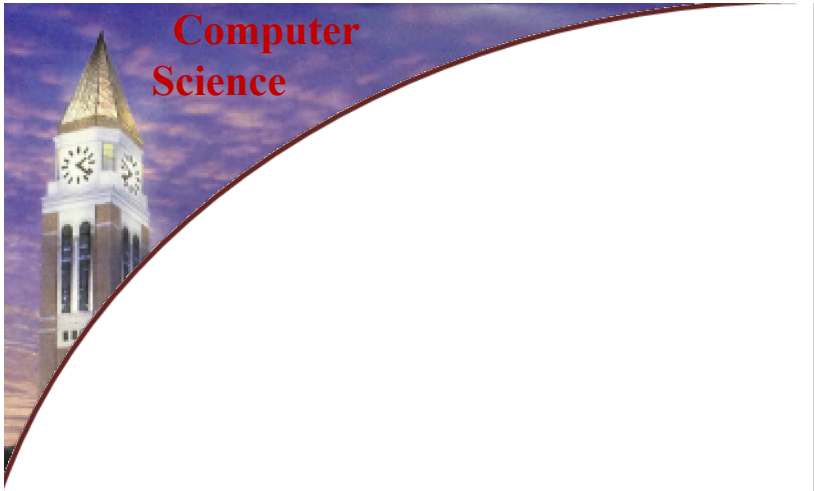
```
(define (apply-env env search-var)
    (cases Env env
      (empty-env () (raise "No such variable found"))
      (extend-env
         (saved-var saved-val saved-env)
            (if (eqv? search-var saved-var)
                 saved-val
                (apply-env saved-env search-var)))))
```



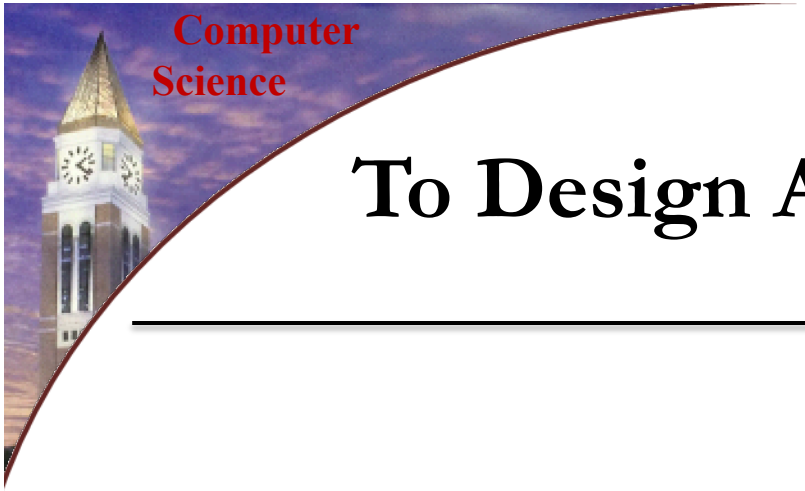

```
(define-datatype Env Env?
   (empty-env)
   (extend-env (var symbol?)(val number?)(env Env?))
)
```

```
(define (apply-env env search-var)
    (cases Env env
      (empty-env () (raise "No such variable found"))
      (extend-env
         (saved-var saved-val saved-env)
           (if (eqv? search-var saved-var)
               saved-val
               (apply-env saved-env search-var)))))
```

pattern matching



HW 5
P2-b



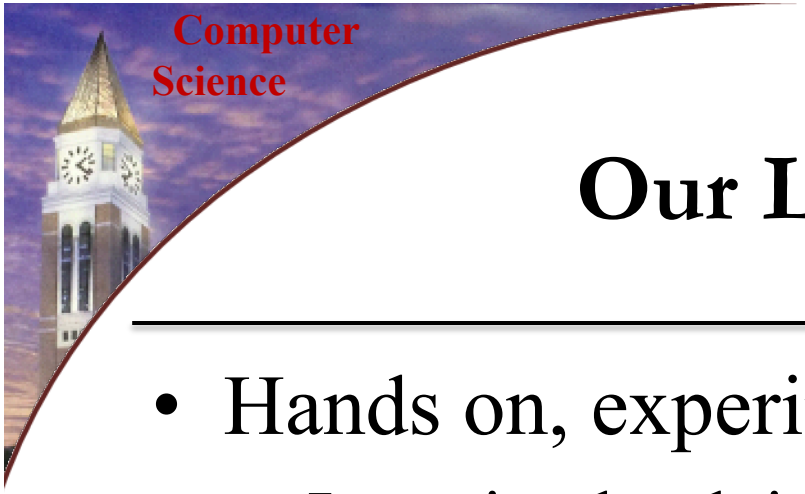
To Design A Programming Language

To design and implement a programming language that has the following features:

- Conditional constructs
- Variable definition and usage
- Procedural definition and procedural call
- Recursive procedure definition and call

To Design A Programming Language

- Garbage collection
- Type checking
- Type inference
- …



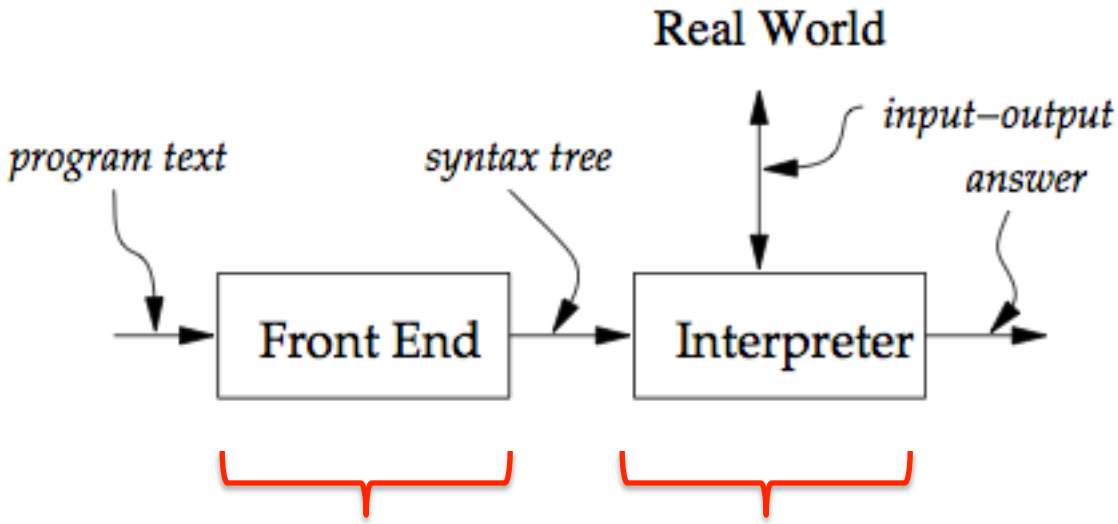
Our Learning Style

- Hands on, experiment-based approach
  - Learning by doing it.

Our goal is to write a program to implement a programming language!

The Basic Form Of The Interpreter

```
(value-of exp env) = val
```

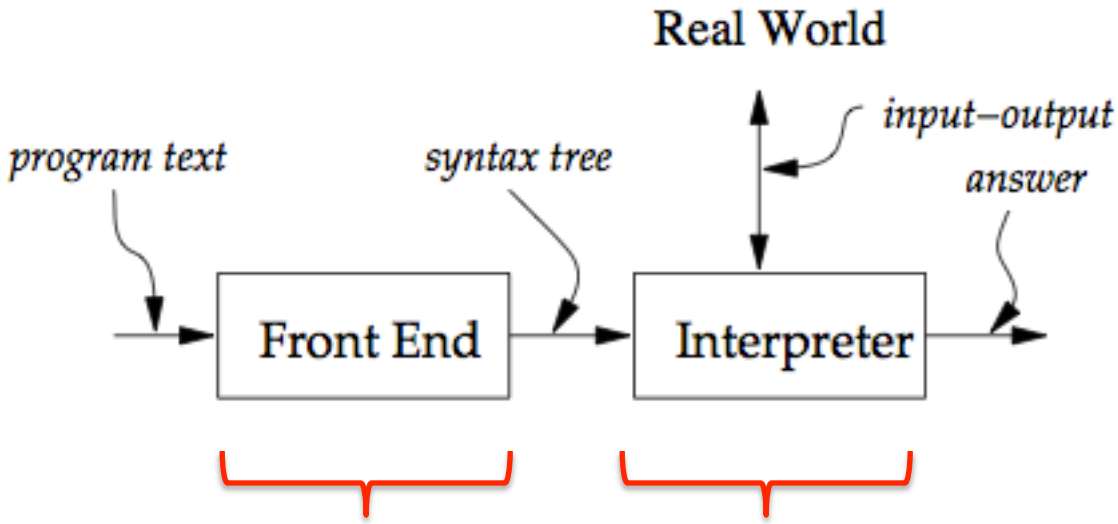


scanning & parsing based on lexical and grammar spec

majority of your work here!

The Basic Form Of The Interpreter

```
(value-of exp env) = val
```

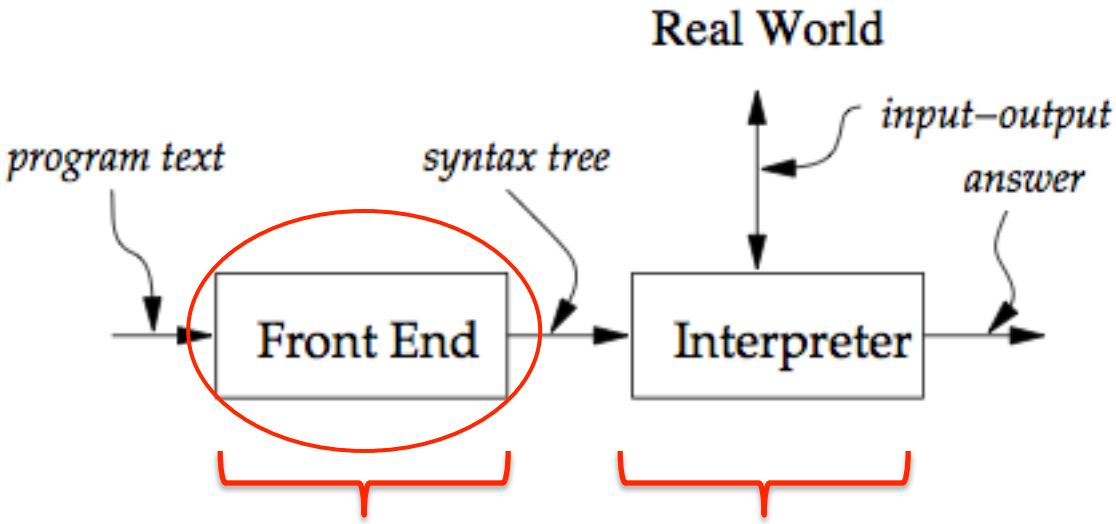


scanning & parsing based on lexical and grammar spec

majority of your work here!

The Basic Form Of The Interpreter

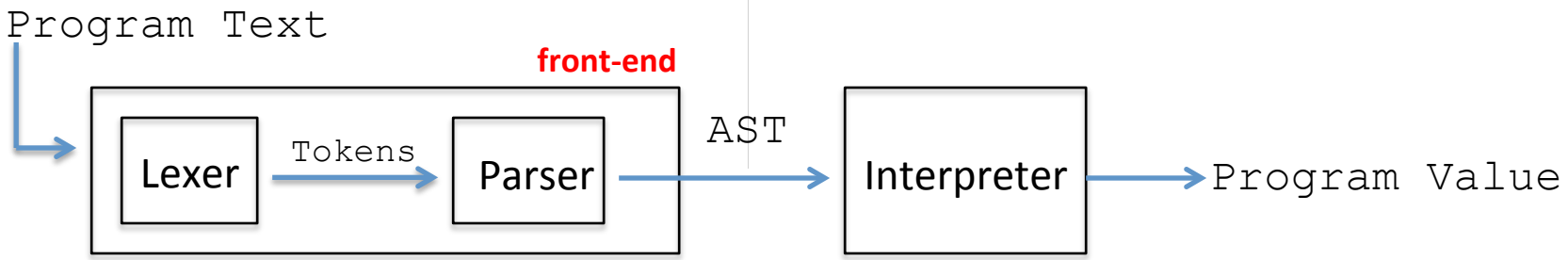
```
(value-of exp env) = val
```



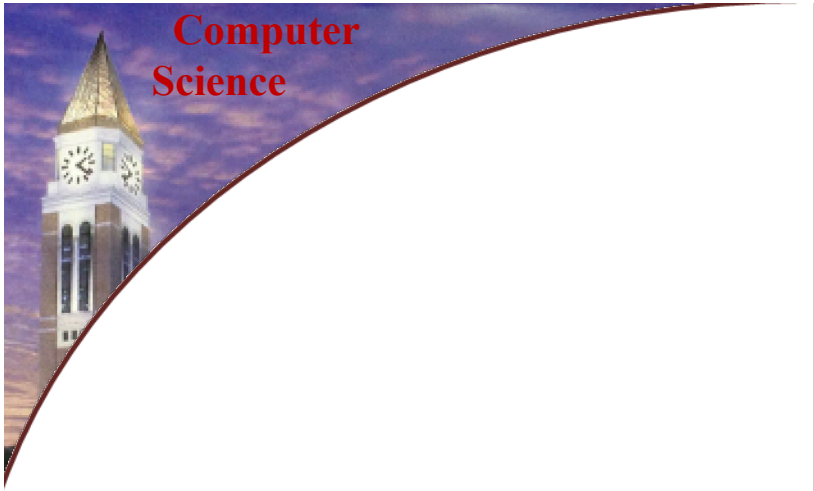
scanning & parsing based on lexical and grammar spec

majority of your work here!

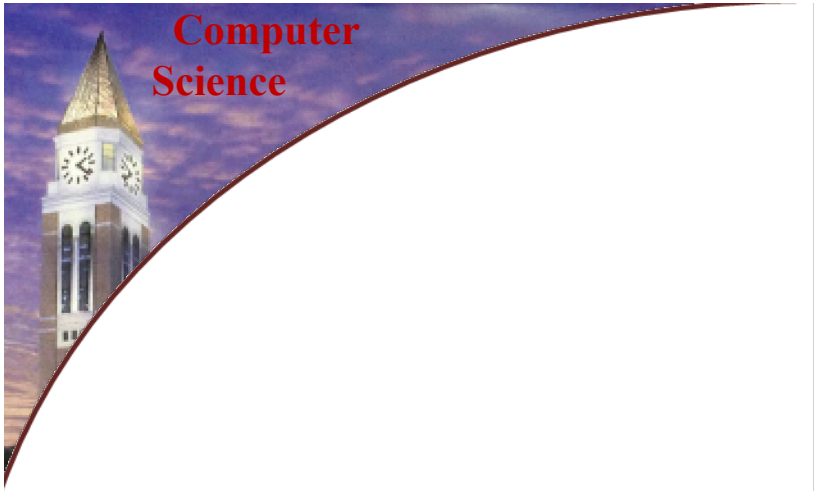
Structural Overview



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



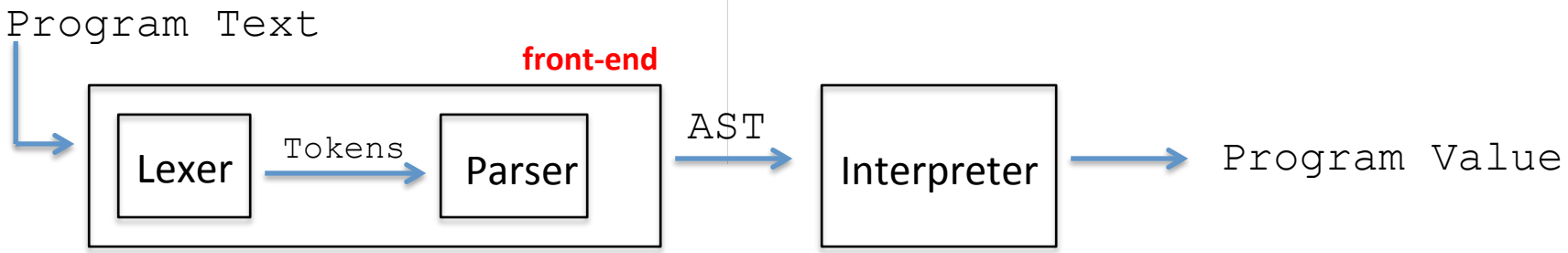
IMPLEMENTING A PROGRAMMING LANGUAGE OF YOUR DESIGN



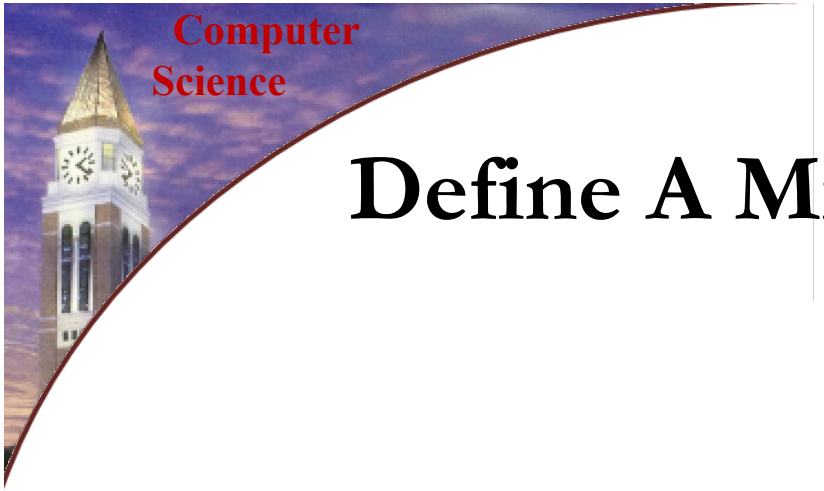
Suggested reading:

- 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

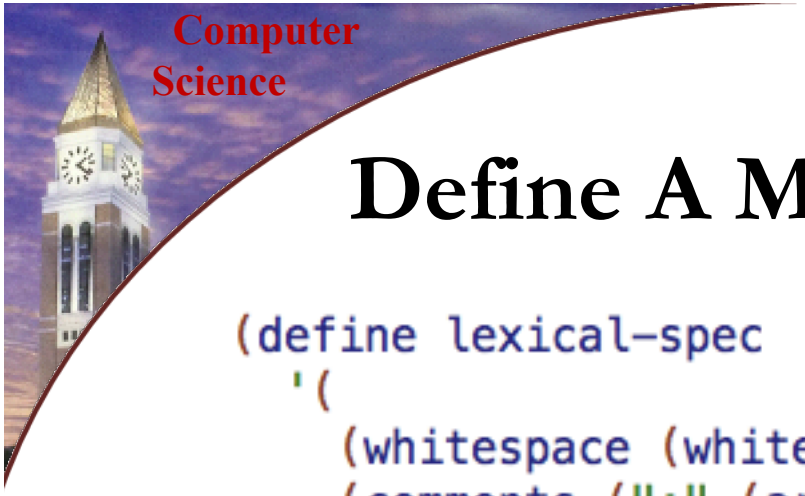


AST: **A**bstract **S**yntax **T**ree



Define A Mini `step` language

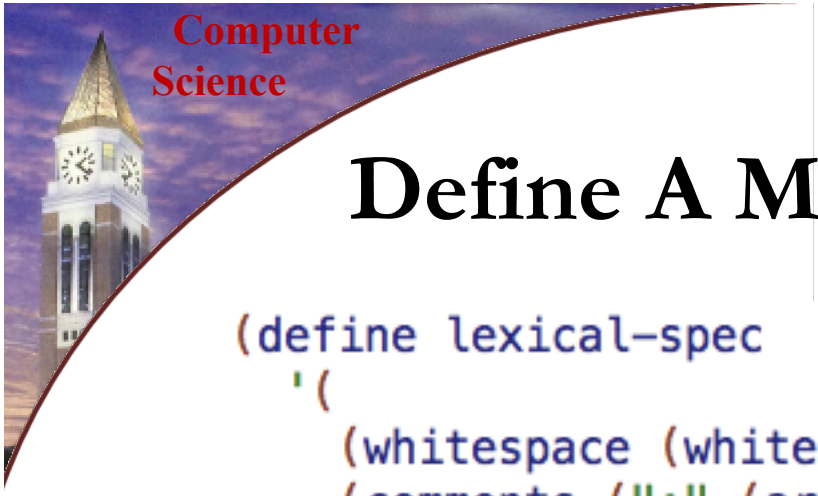
first: define the tokens
(lexical specification)



Define A Mini `step` language

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

first: define the tokens (lexical specification)

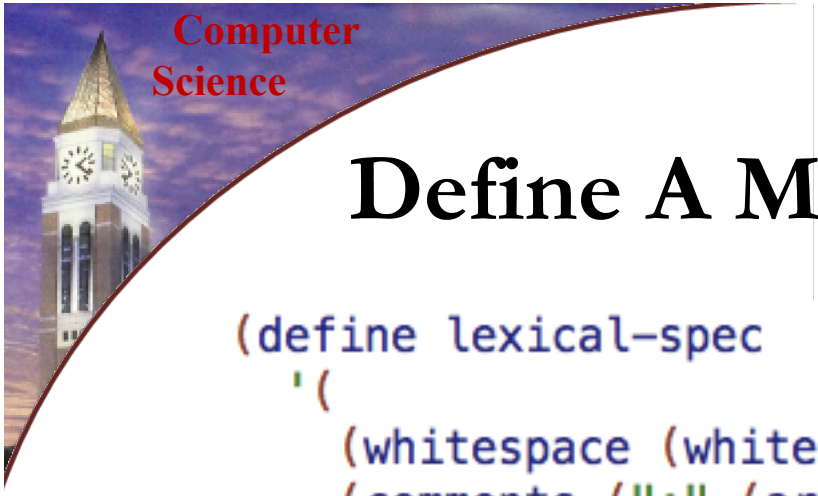


Define A Mini `step` language

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

first: define the tokens (lexical specification)

then: define the grammar (grammar specification, where tokens are used)

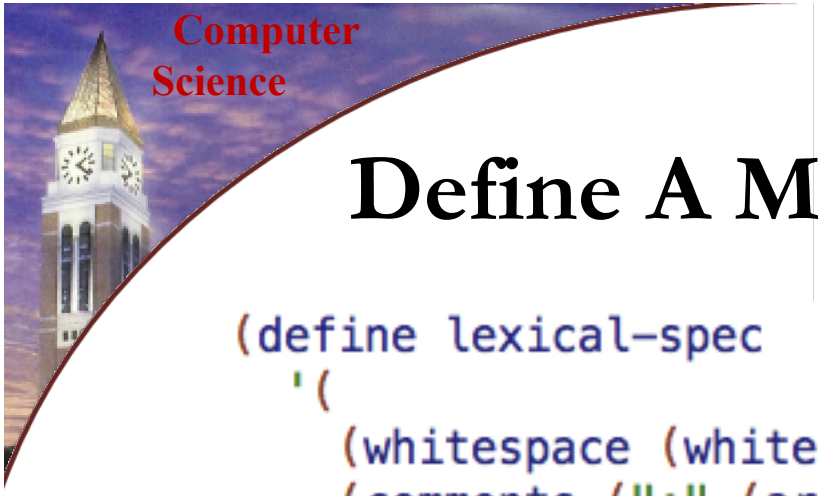


Define A Mini `step` language

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

(define grammar-spec
  '(
    (program (step) a-program)
    (step ("left" num) left-step)
    (step ("right" num) right-step)
    (step ("(" step step ")") seq-step)))
```

first: define the tokens (lexical specification)

then: define the grammar (grammar specification, where tokens are used)



Define A Mini `step` language

token actions

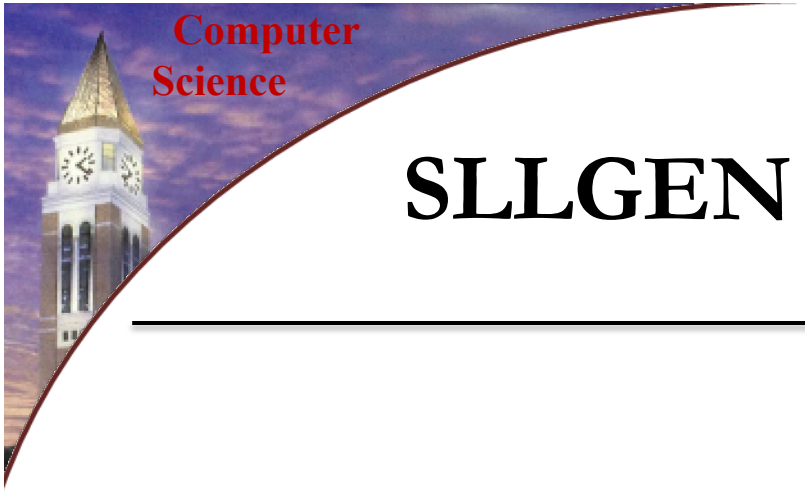
```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comments (";" (arbno (not #\newline))) skip)
    (num       (digit (arbno digit) )   number)
    )
  )

(define grammar-spec
  '(
    (program (step) a-program)
    (step ("left" num) left-step)
    (step ("right" num) right-step)
    (step ("(" step step ")") seq-step)))
```

Define A Mini `step` language

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

(define grammar-spec
  '(
    (program (step) a-program)
    (step ("left" num) left-step)
    (step ("right" num) right-step)
    (step ("(" step step ")") seq-step)))
```

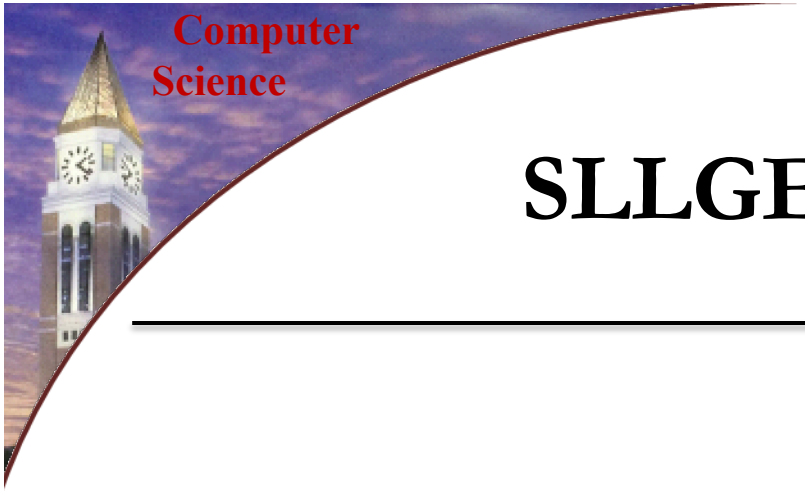
token actions

tokens are used in your grammar



SLLGEN Boiler Plate Code

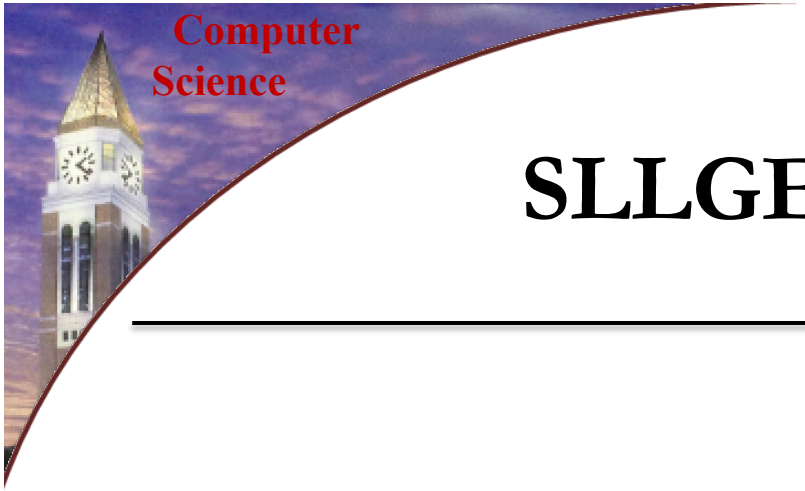
```
(sllgen:make-define-datatypes lexical-spec grammar-spec)

(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

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

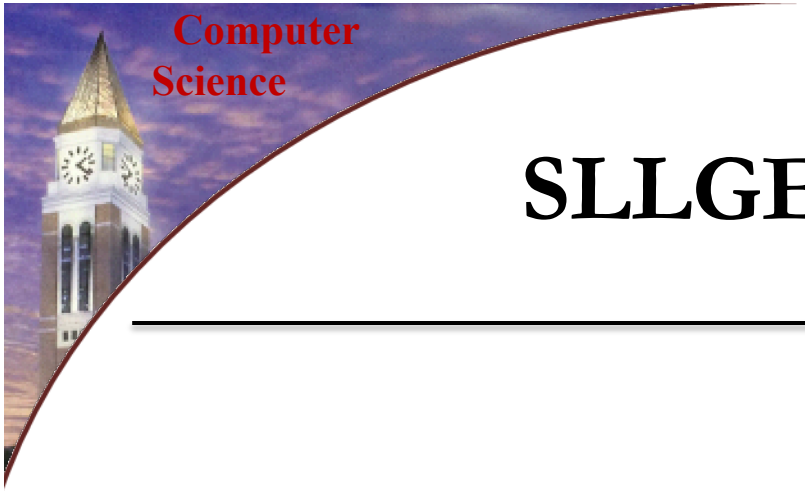
This will create the `Abstract Syntax Tree` using `define-datatype` based on a given **lexical specification** and a **grammar specification**.



SLLGEN Boiler Plate

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

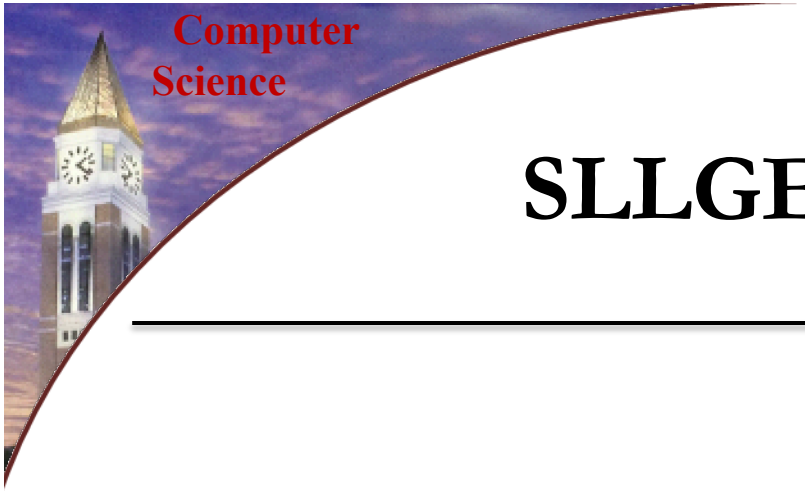
This will create the `Abstract Syntax Tree` using `define-datatype` based on a given **lexical specification** and a **grammar specification**.



SLLGEN Boiler Plate

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

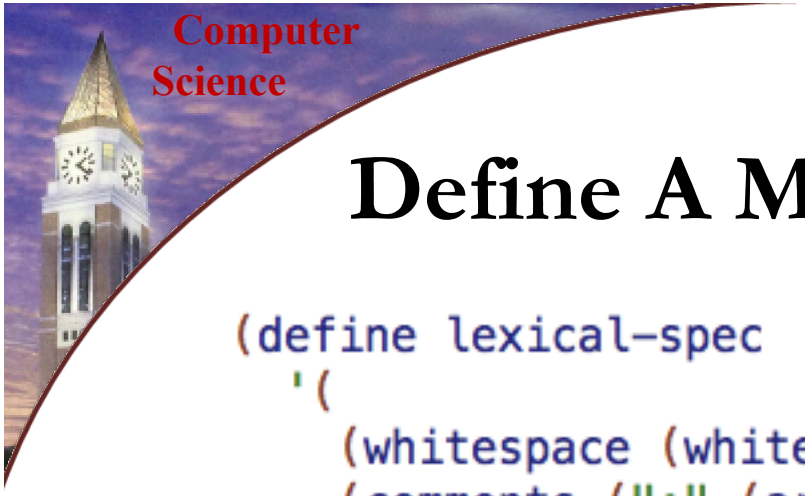
define the Abstract Syntax Tree as the return value of **(show-data-types)** function.



SLLGEN Boiler Plate

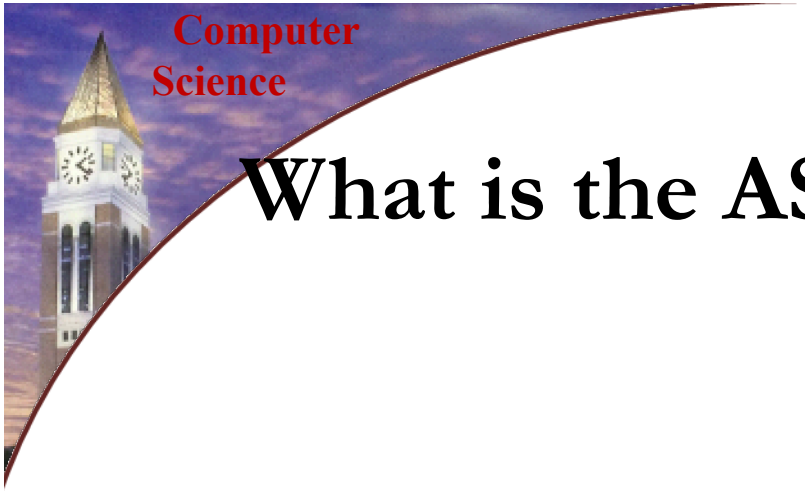
`parser` is a **one argument function** that takes a string, scans & parses it and generates the corresponding **Abstract Syntax Tree**.

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



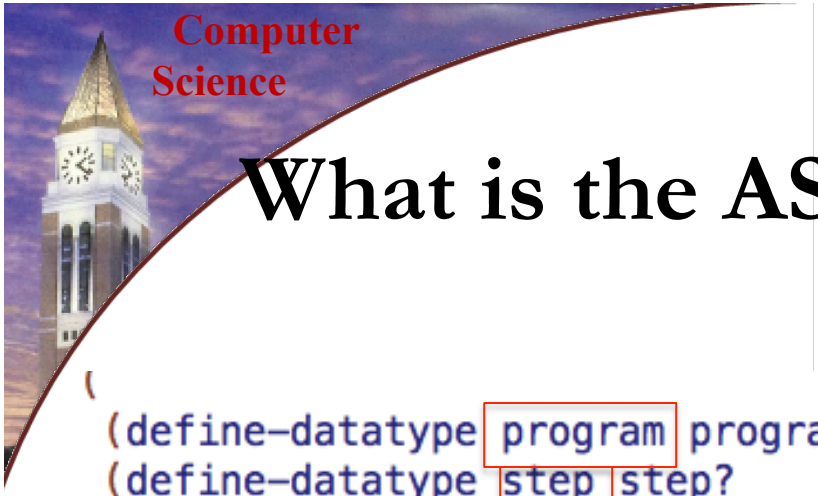
Define A Mini `step` language

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

(define grammar-spec
  '(
    (program (step) a-program)
    (step ("left" num) left-step)
    (step ("right" num) right-step)
    (step ("(" step step ")") seq-step)))
```



What is the AST for `step` language

Use the `(show-data-types)` boiler plate code seen on **slide 35** to find it out -



What is the AST for `step` language

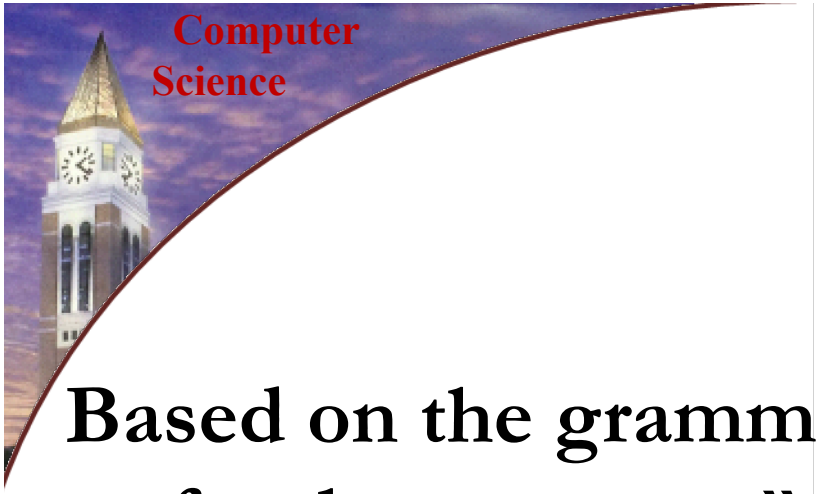
```
(
 (define-datatype program program? (a-program (a-program6 step?)))
 (define-datatype step step?
   (left-step (left-step7 number?))
   (right-step (right-step8 number?))
   (seq-step (seq-step9 step?) (seq-step10 step?)))
 )
```

Compare it with the grammar specification on slide 29

What is the overall AST for `step` language

```
(
 (define-datatype program program? (a-program (a-program6 step?)))
 (define-datatype step step?
   (left-step (left-step7 number?))
   (right-step (right-step8 number?))
   (seq-step (seq-step9 step?) (seq-step10 step?)))
 )
```

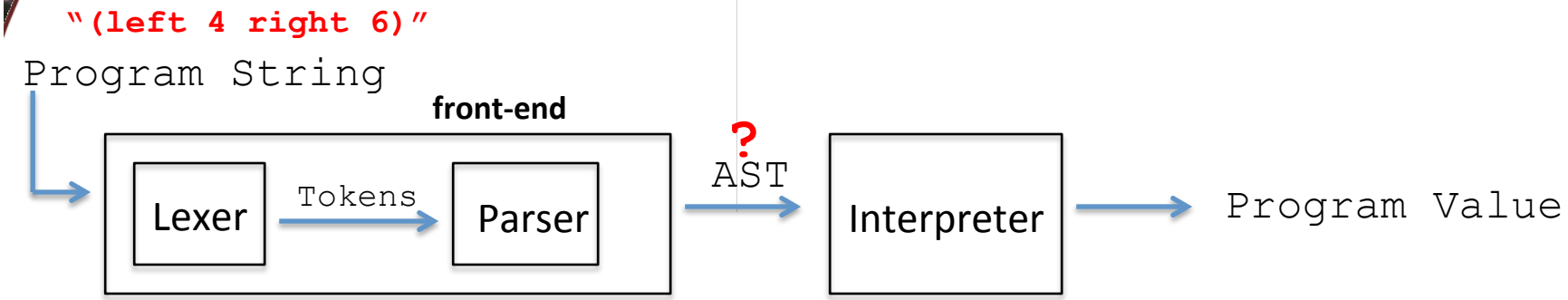
```
(define grammar-spec
  '(
    (program (step) a-program)
    (step ("left" num) left-step)
    (step ("right" num) right-step)
    (step ("(" step step ")") seq-step)))
```



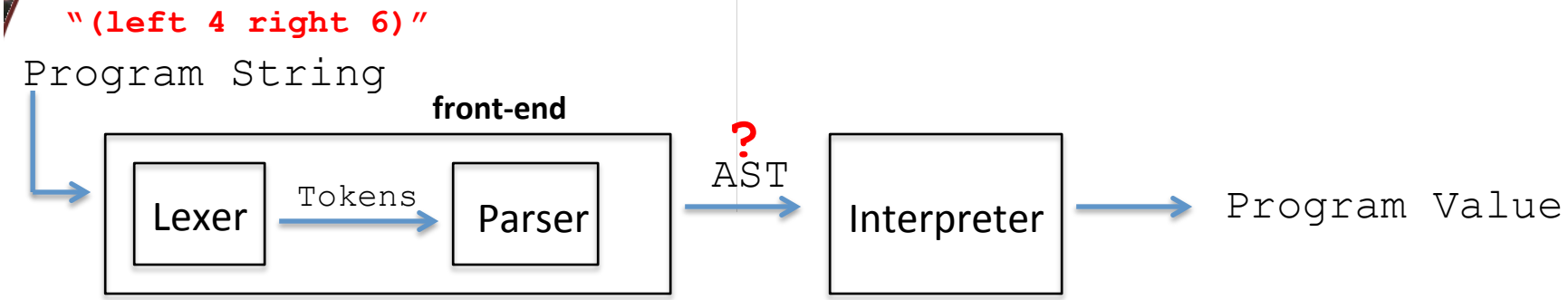
Based on the grammar, what is the specific AST for the program "`(left 4 right 6)`"?

Use the `parser` boiler plate code seen on **slide 36** to find it out -

Structural Overview Of P/L Processing Systems

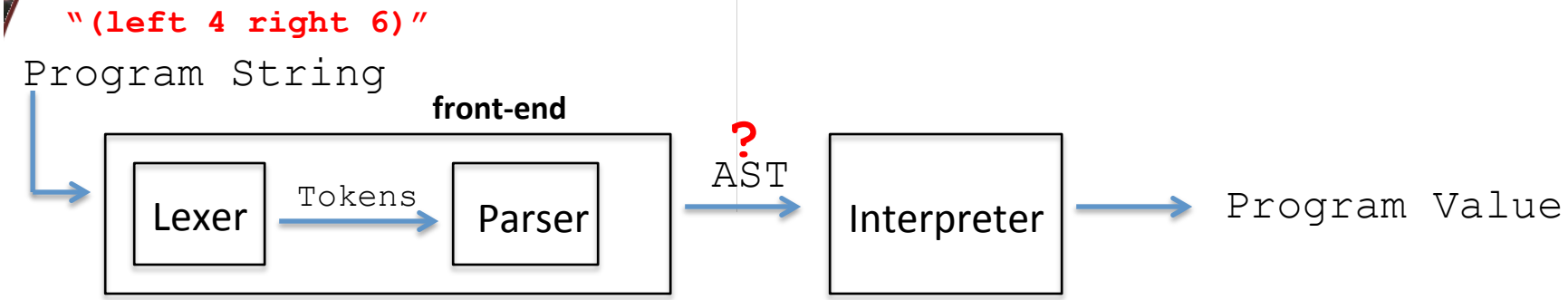


Structural Overview Of P/L Processing Systems



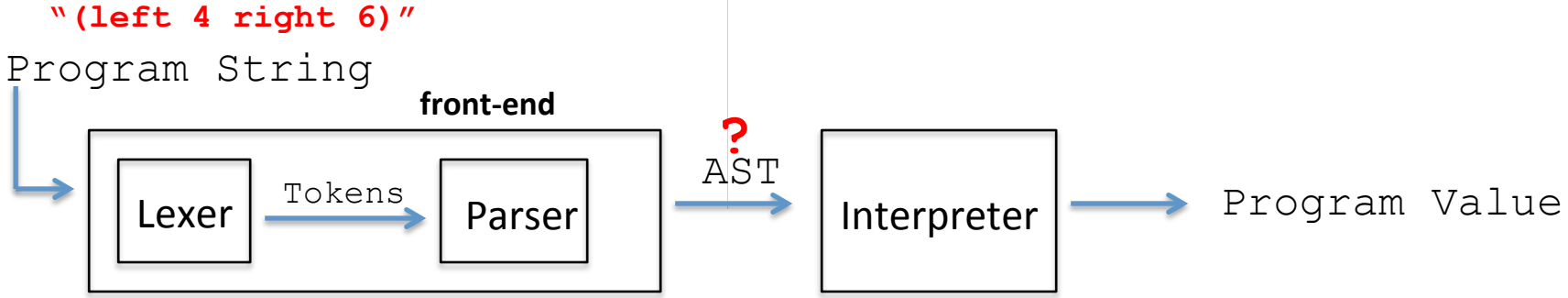

```
> (parser "(left 4 right 6)")
```

Structural Overview Of P/L Processing Systems



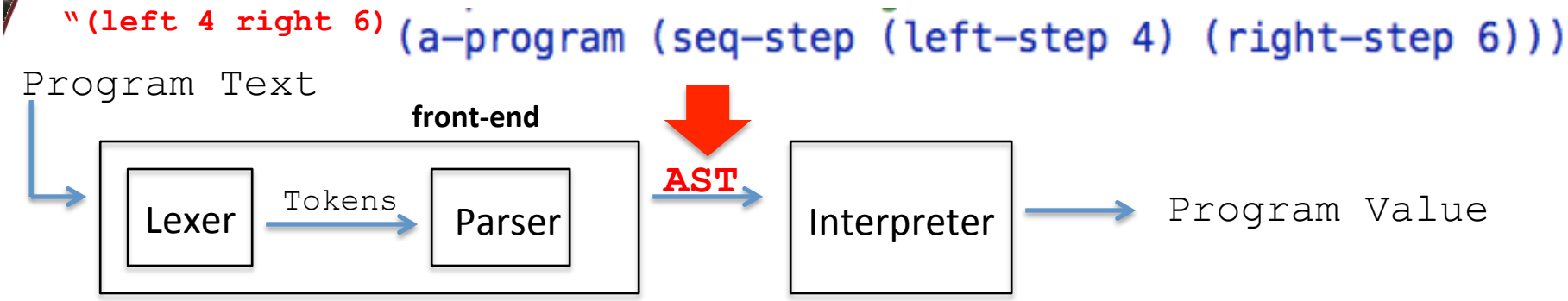

```
> (parser "(left 4 right 6)")
(a-program (seq-step (left-step 4) (right-step 6)))
```

Structural Overview Of P/L Processing Systems

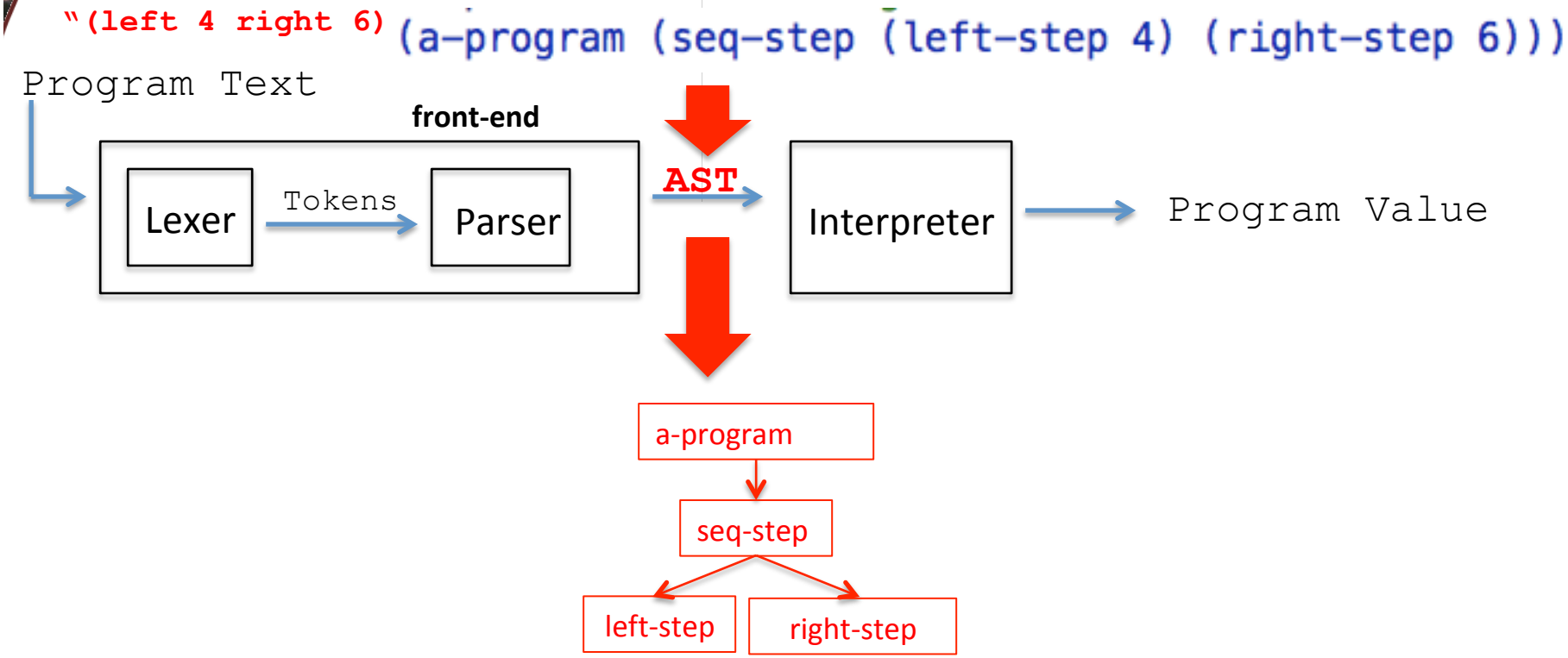



```
> (parser "(left 4 right 6)")
(a-program (seq-step (left-step 4) (right-step 6)))
```

Internal Representation of Program Values



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