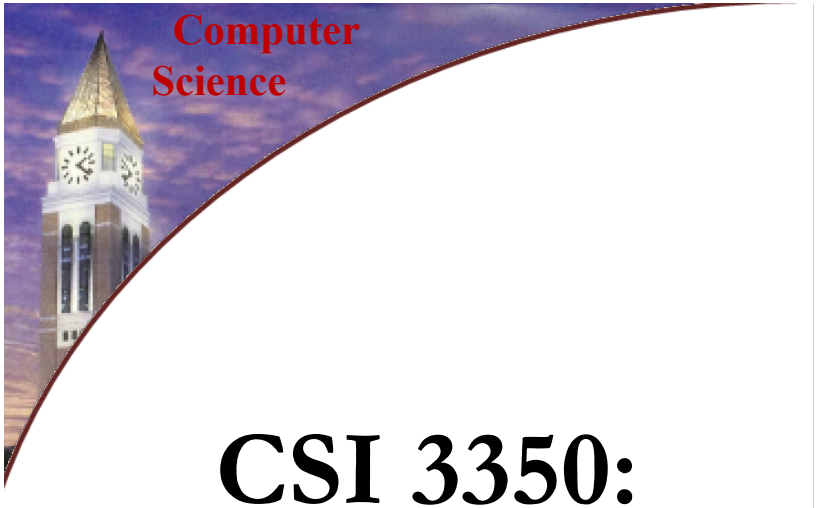


CSI 3350: PROGRAMMING LANGUAGES

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 !

JavaScript, Python are the Most popular programming languages now !

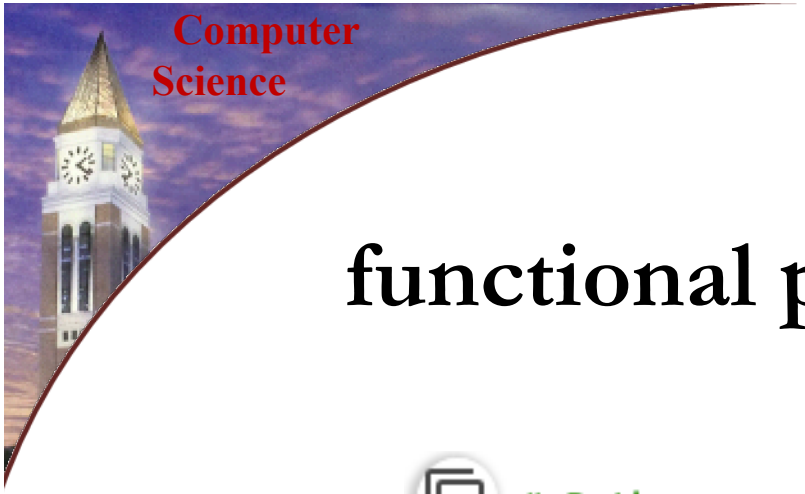
functional programming in Python !

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
```

JavaScript, Python are the Most popular programming languages now !

functional programming in Python !

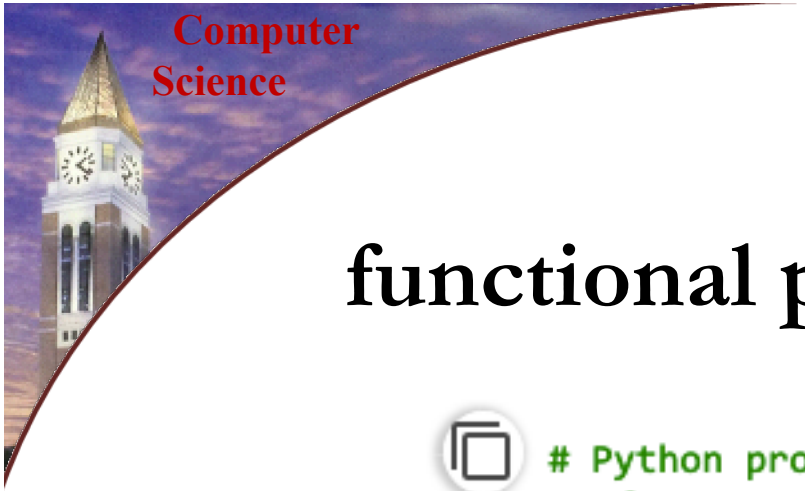
```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Output: 24
```



functional programming in Python !

```
# Python program to demonstrate working
# of map.

# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Output :

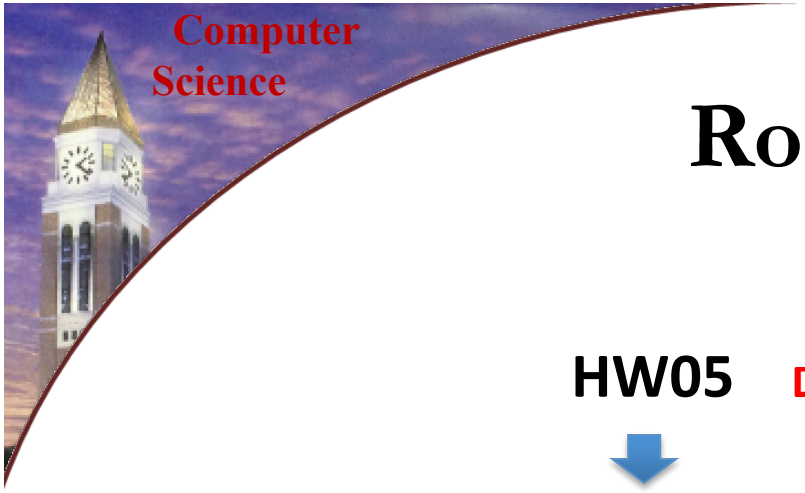


functional programming in Python !

```python
# Python program to demonstrate working
# of map.

# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Output :

```
{2, 4, 6, 8}
```



Road Ahead -

HW05 Due: Nov 12

↓

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

↓

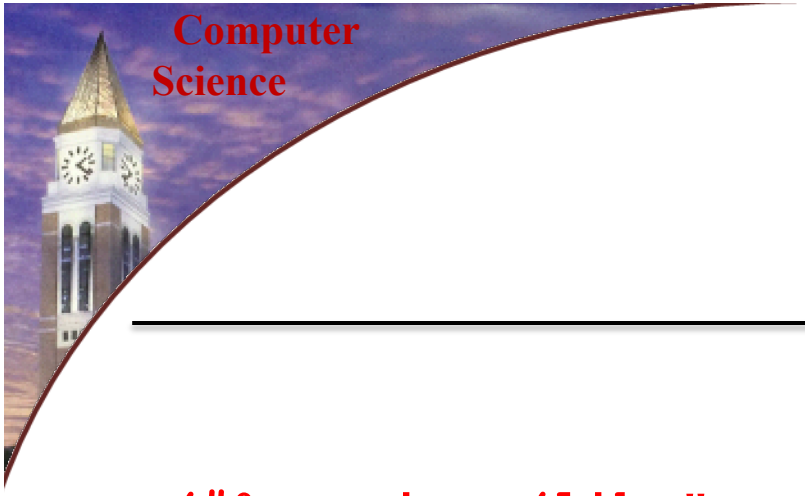
Exam02 ← **Nov 27**

↓

HW07

↓

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

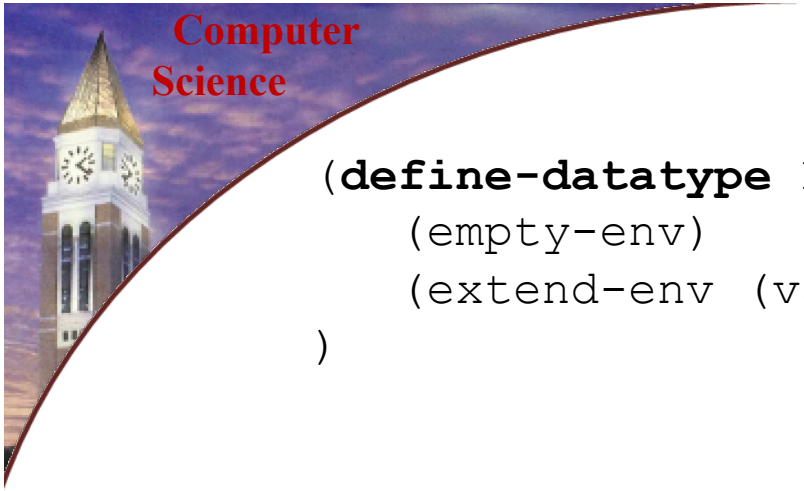



```
(#%require (lib "eopl.ss" "eopl"))
```

define-datatype

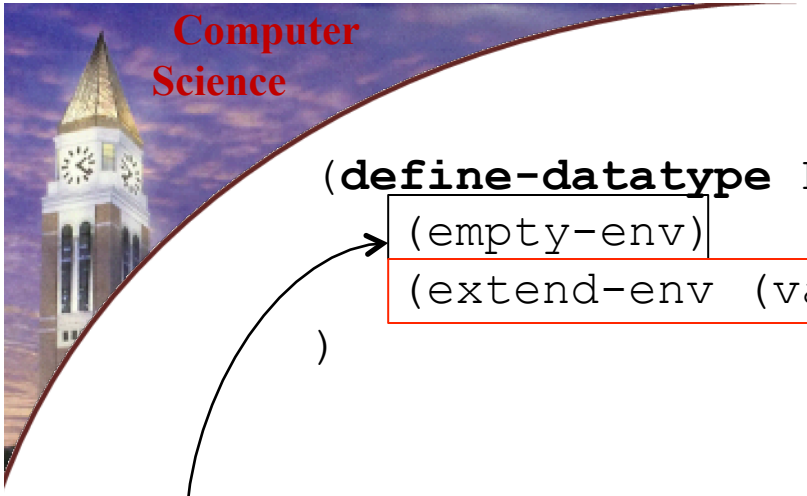
`cases` Syntax Abstraction

`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)))))
```



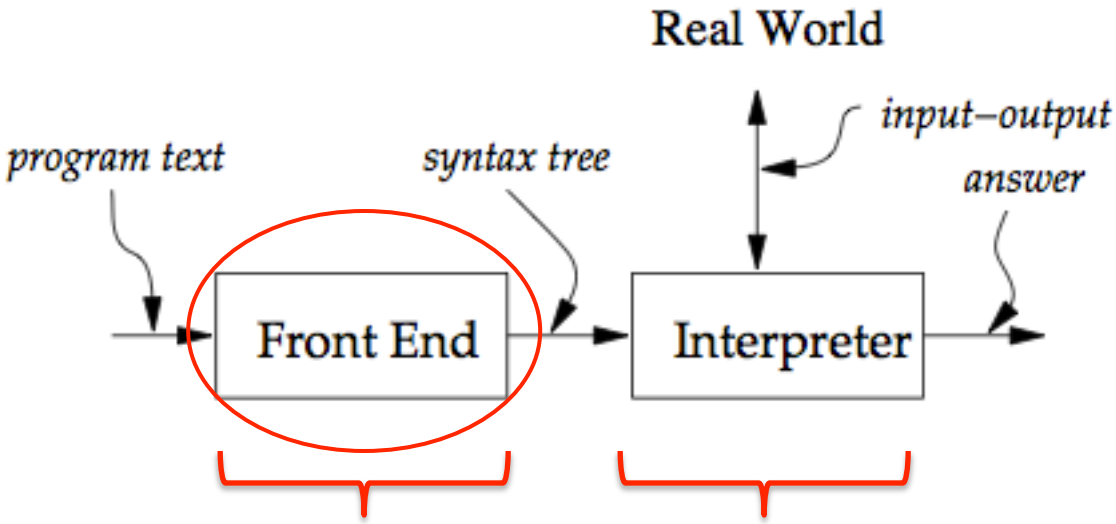

```
(define-datatype Env Env?
   (empty-env)
   (extend-env (var symbol?)(val number?)(env Env?))
)
```

```
(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)))))
```

pattern matching

The Basic Form Of The Interpreter

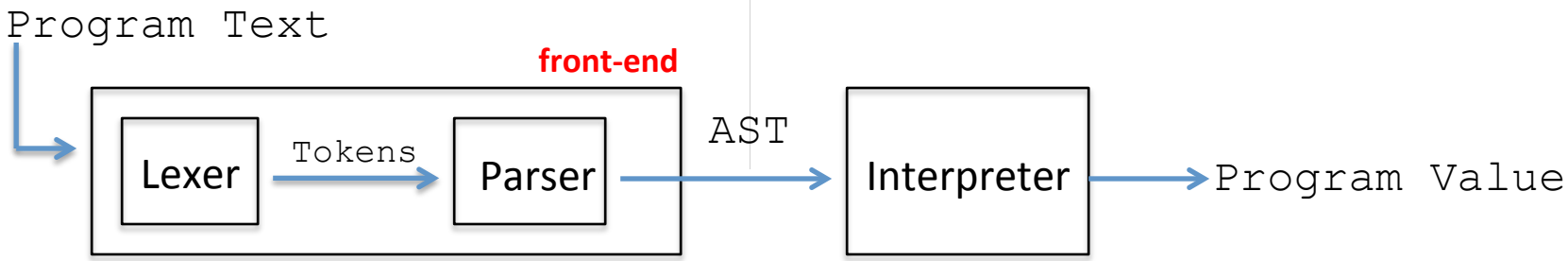
```
(value-of exp env) = val
```



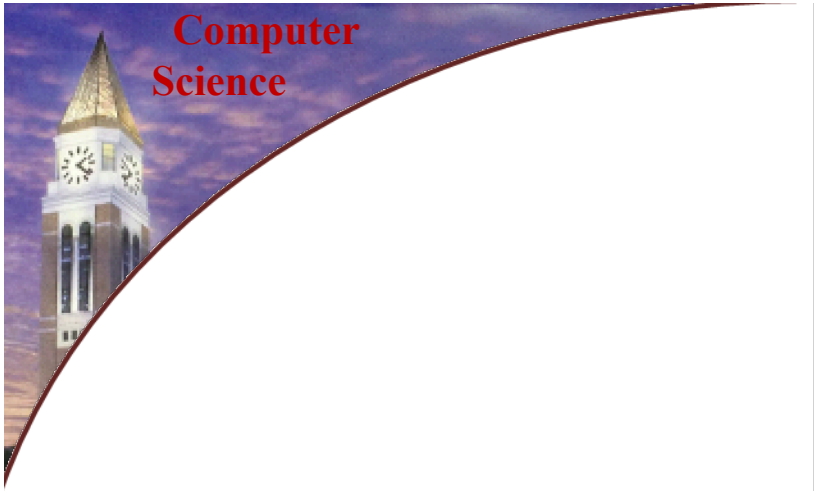
scanning & parsing based on lexical and grammar spec

majority of your work here!

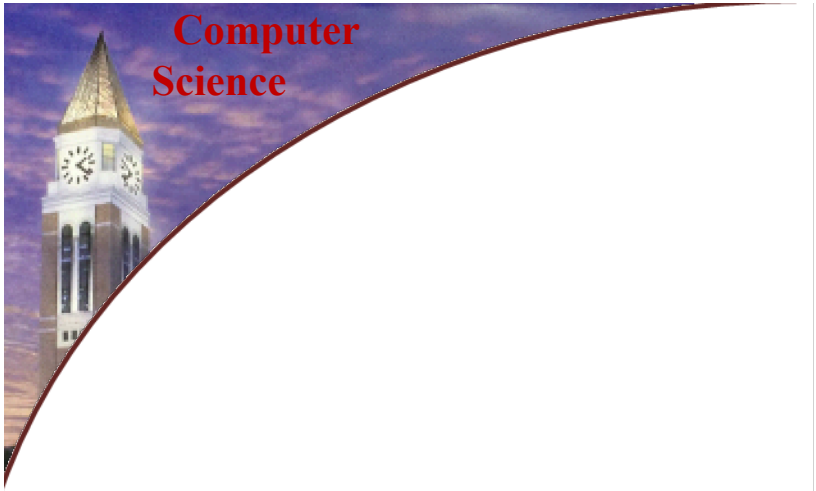
Structural Overview



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



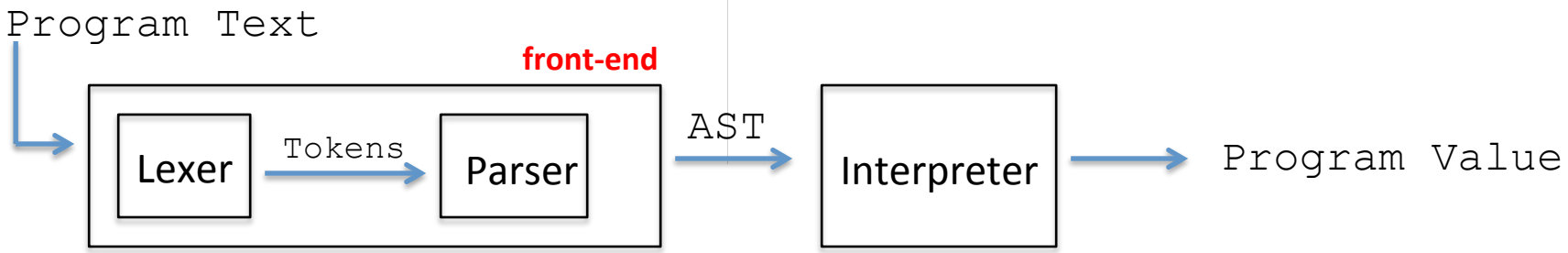
IMPLEMENTING A PROGRAMMING LANGUAGE OF YOUR DESIGN



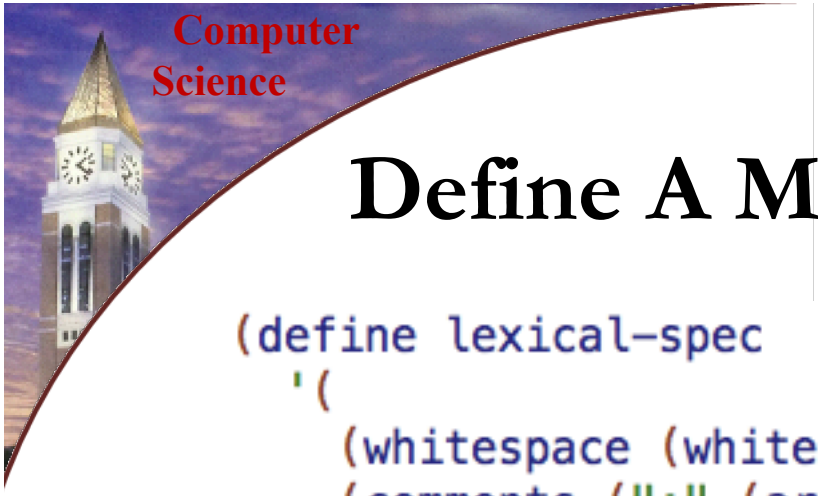
Suggested reading:

- 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

```
(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

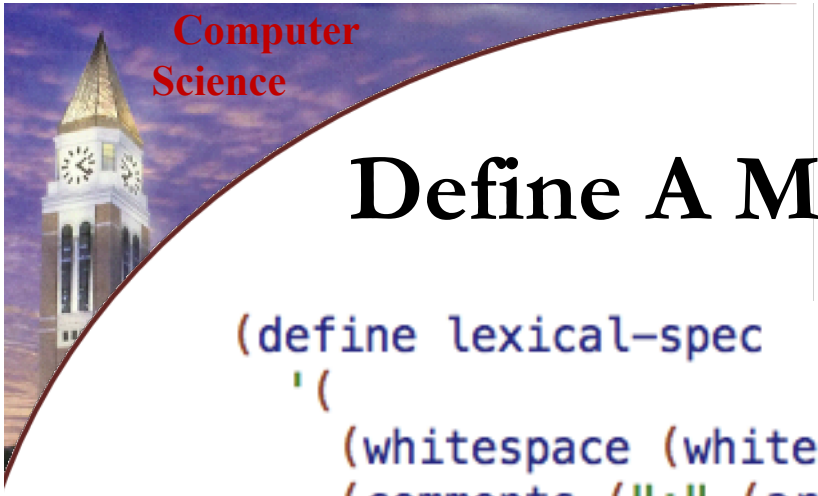
```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comments (";" (arbno (not #\newline))) skip)
    (num       (digit (arbno digit) )   number)
    )
  )

(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

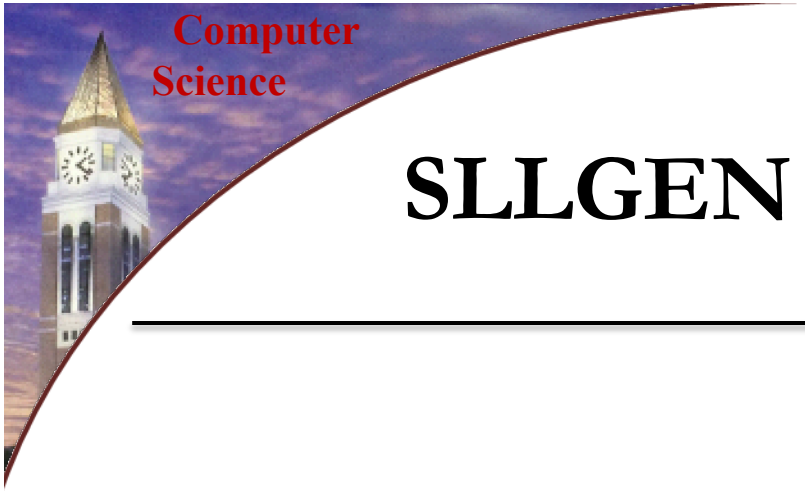


Define A Mini `step` language

```
(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)))
```

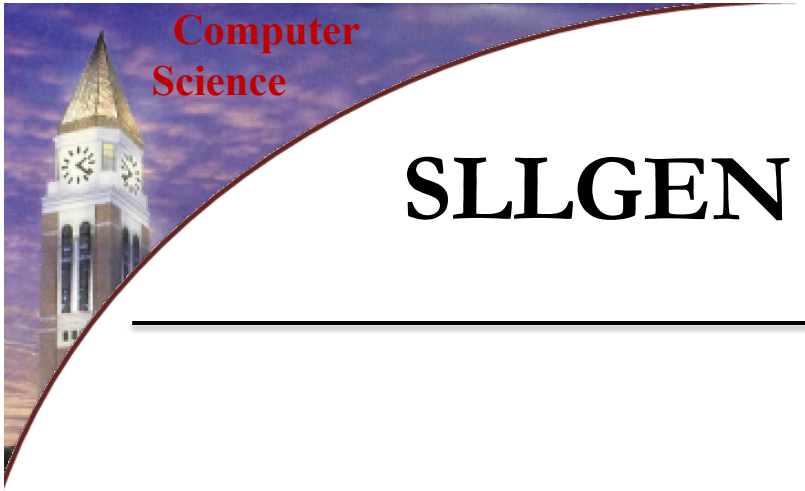
first: define the tokens (lexical specification)

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



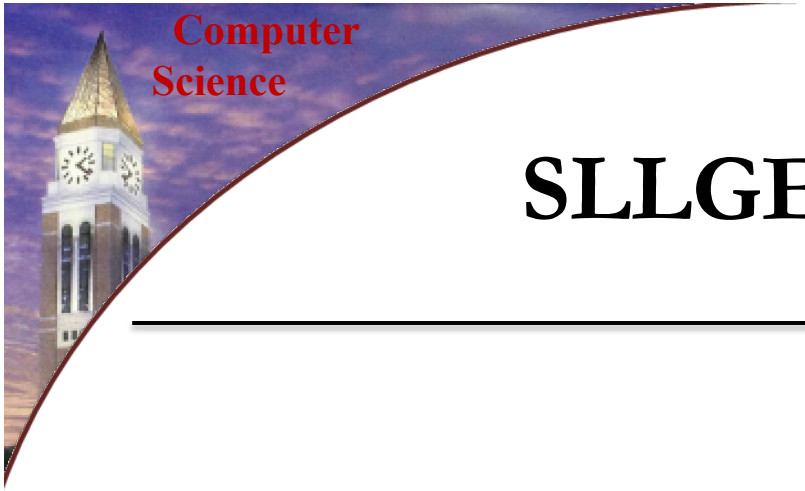
SLLGEN Boiler Plate Code

```
(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 Code

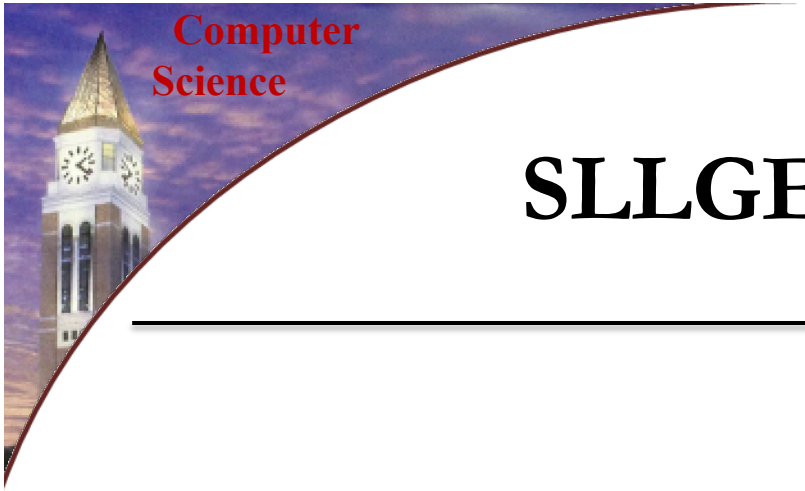
```
(sllgen:make-define-datatypes lexical-spec grammar-spec)

(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)
```

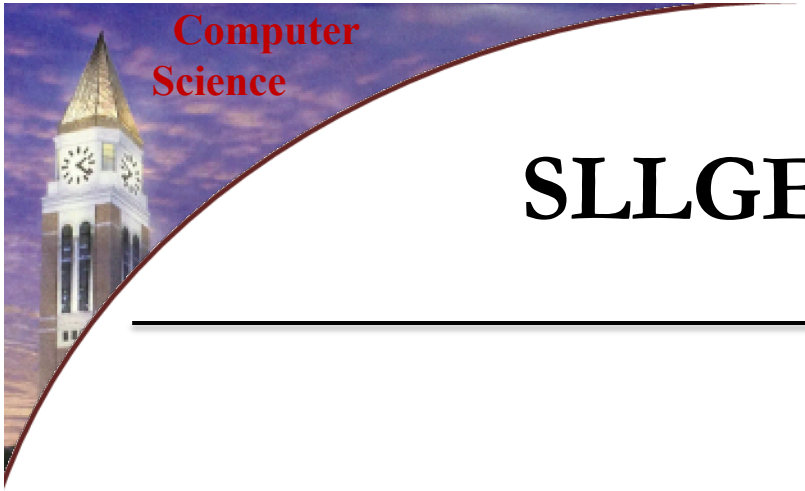
This will create the `Abstract Syntax Tree` using `define-datatype` based on a given **lexical specification** and a **grammar specification**.



SLLGEN Boiler Plate

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

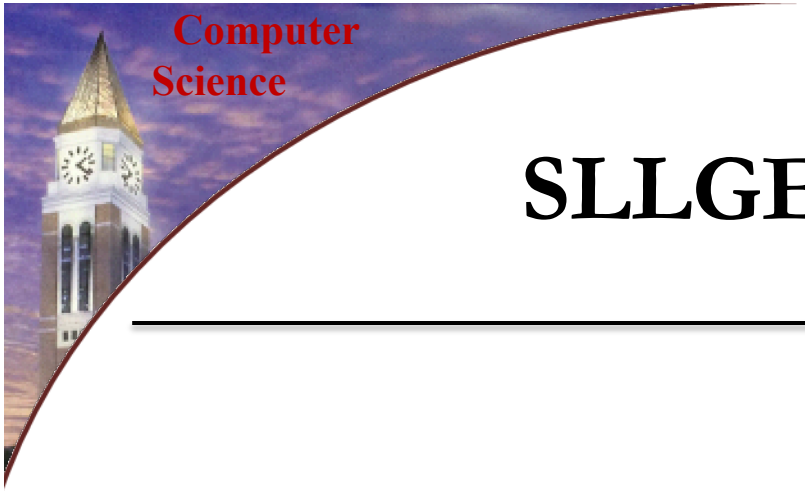
This will create the `Abstract Syntax Tree` using `define-datatype` based on a given **lexical specification** and a **grammar specification**.



SLLGEN Boiler Plate

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

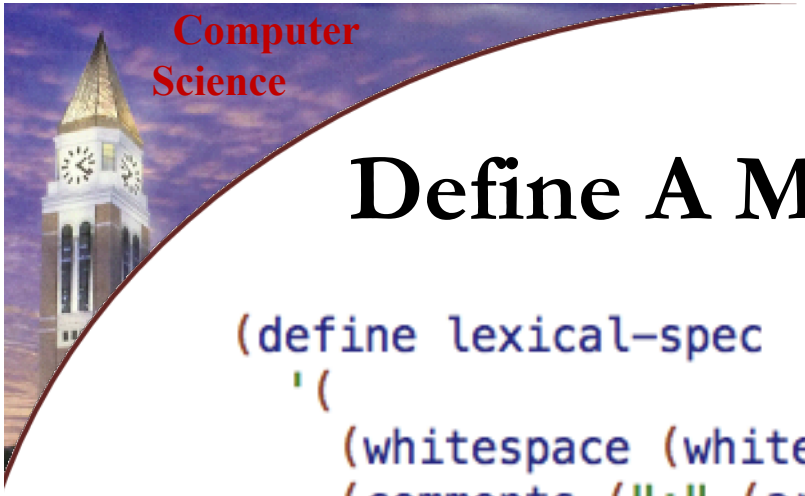
define the Abstract Syntax Tree for the **grammar** as the return value of **(show-data-types)** function.



SLLGEN Boiler Plate

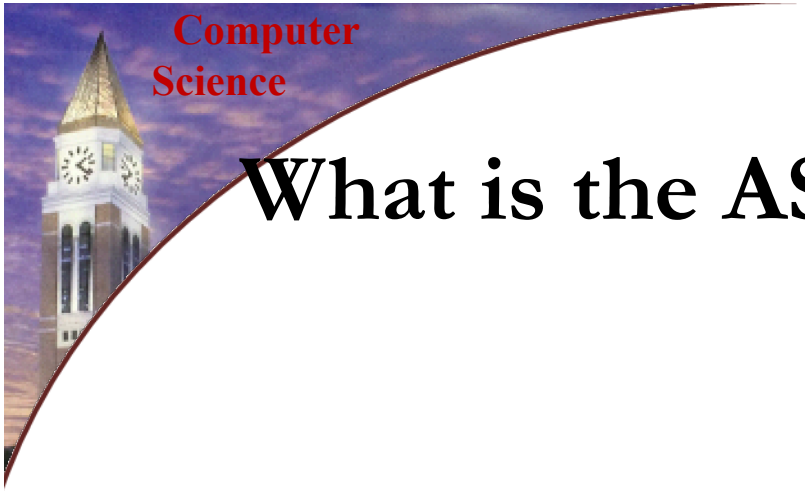
`parser` is a **one argument function** that takes a program string, scans & parses it and generates the corresponding **Abstract Syntax Tree**.

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



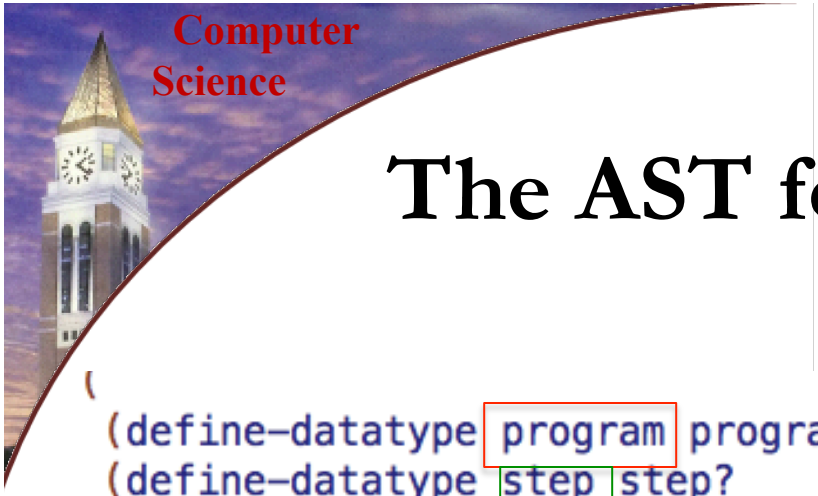
Define A Mini `step` language

```
(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

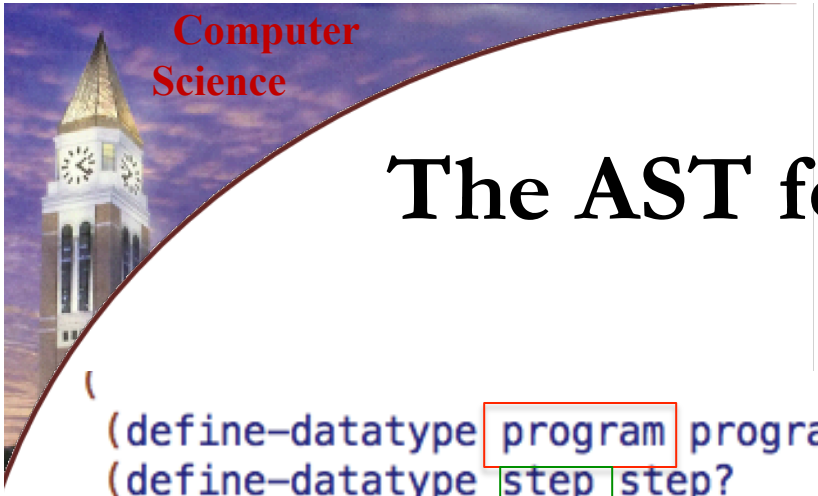
Use the `(show-data-types)` boiler plate code seen on **slide 26** to find it out -



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 below

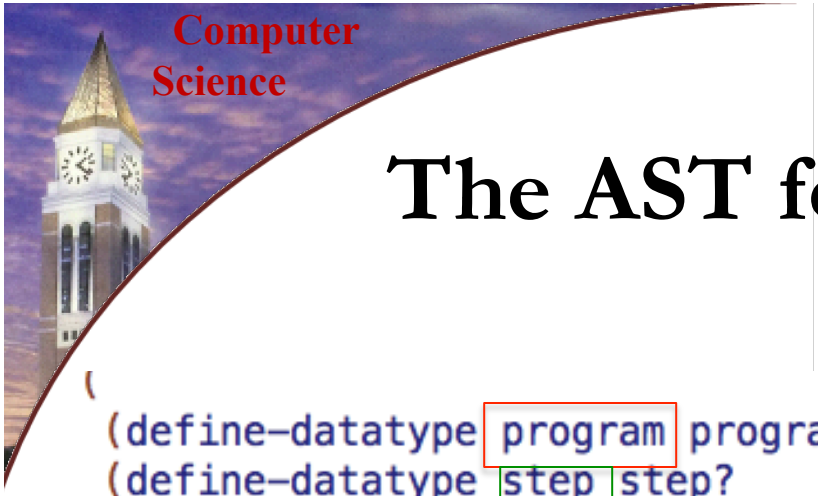


The AST for `step` language

```
(
 (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 below

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

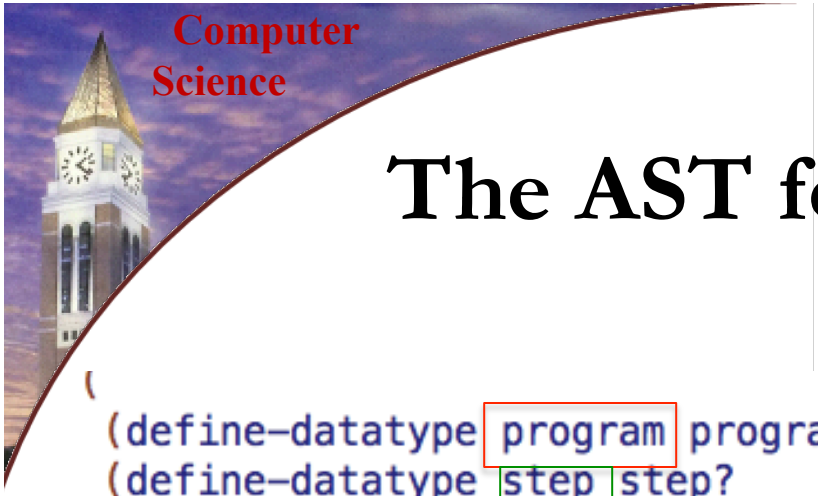


The AST for `step` language

```
(
 (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 below

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



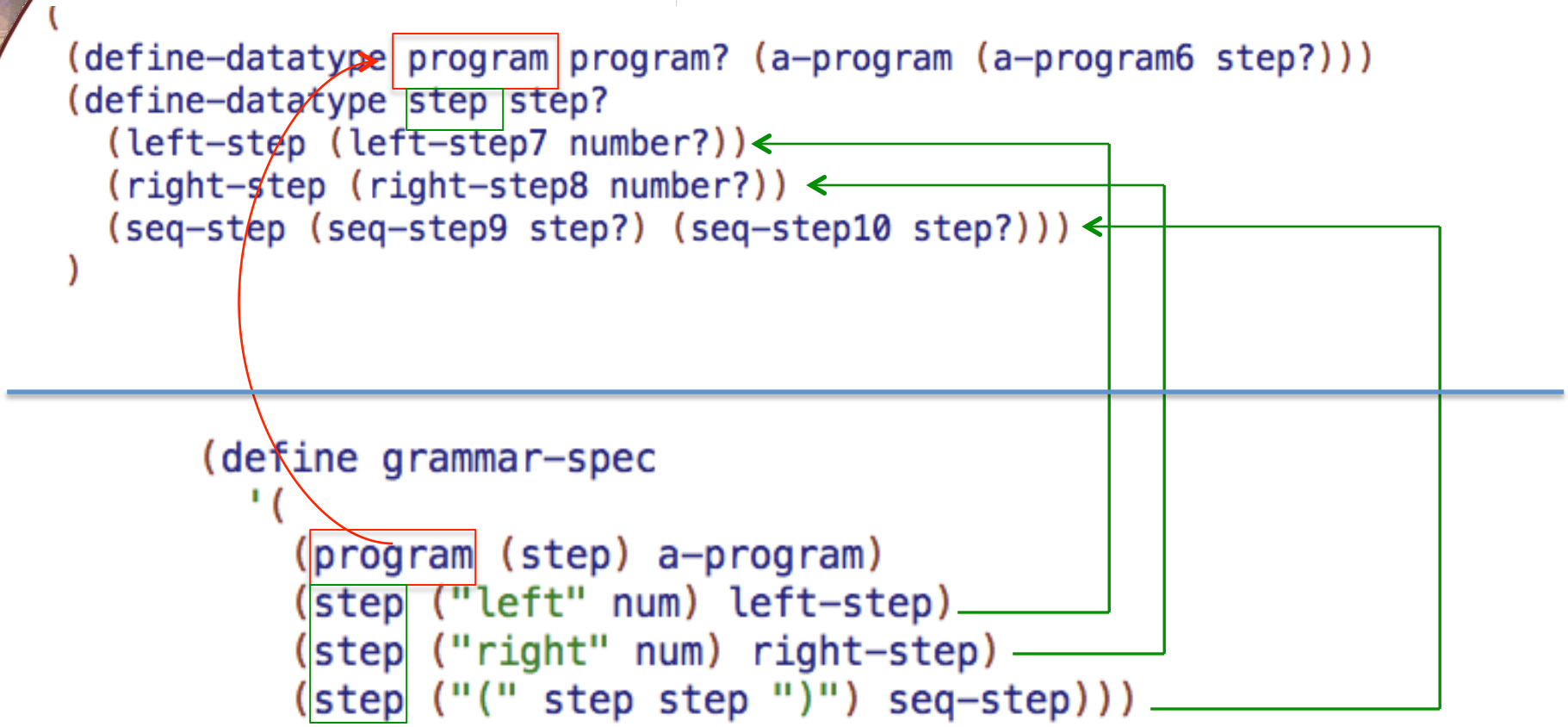
The AST for `step` language

```
(
 (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 below

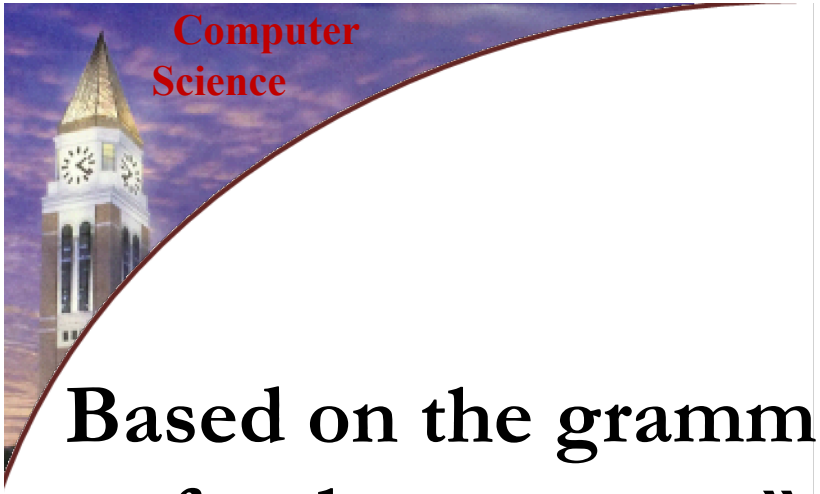
```
(define grammar-spec
  '(
    (program (step) a-program)
    (step ("left" num) left-step)
    (step ("right" num) right-step)
    (step ("(" step step ")") seq-step)))
```

What is the 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?)))
 )
```

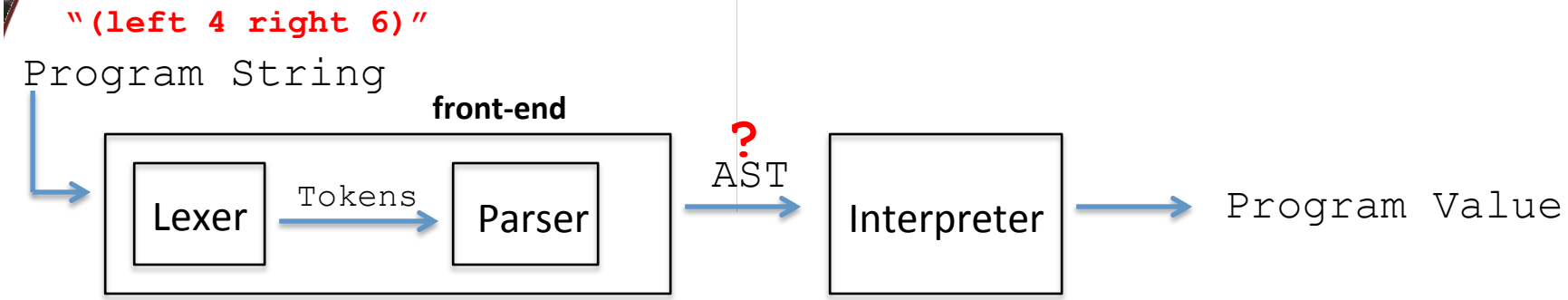
```
(define grammar-spec
  '(
    (program (step) a-program)
    (step ("left" num) left-step)
    (step ("right" num) right-step)
    (step ("(" step step ")") seq-step)))
```



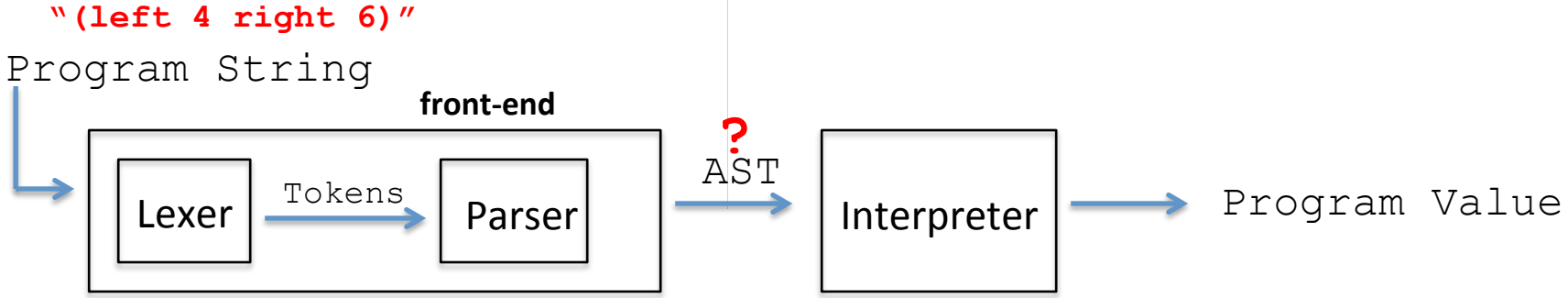
Based on the grammar, what is the specific AST for the program "`(left 4 right 6)`"?

Use the `parser` boiler plate code seen on **slide 24** to find it out -

Structural Overview Of P/L Processing Systems

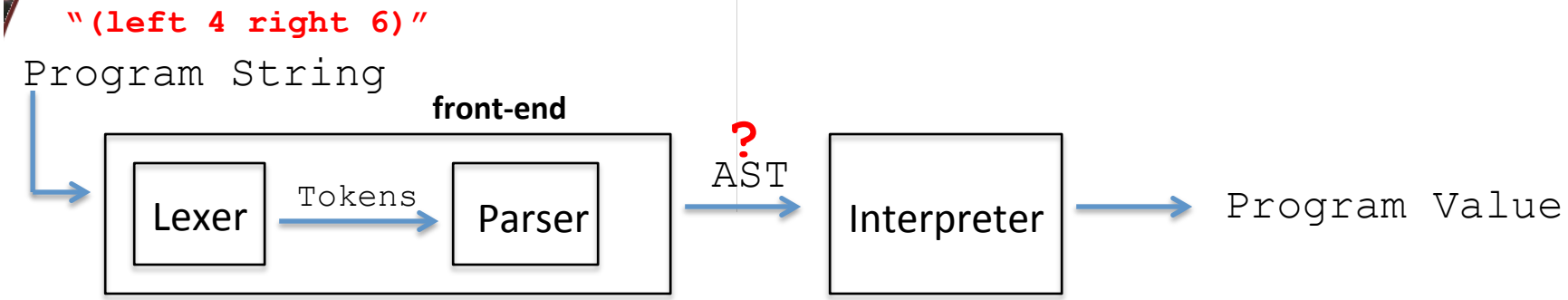


Structural Overview Of P/L Processing Systems



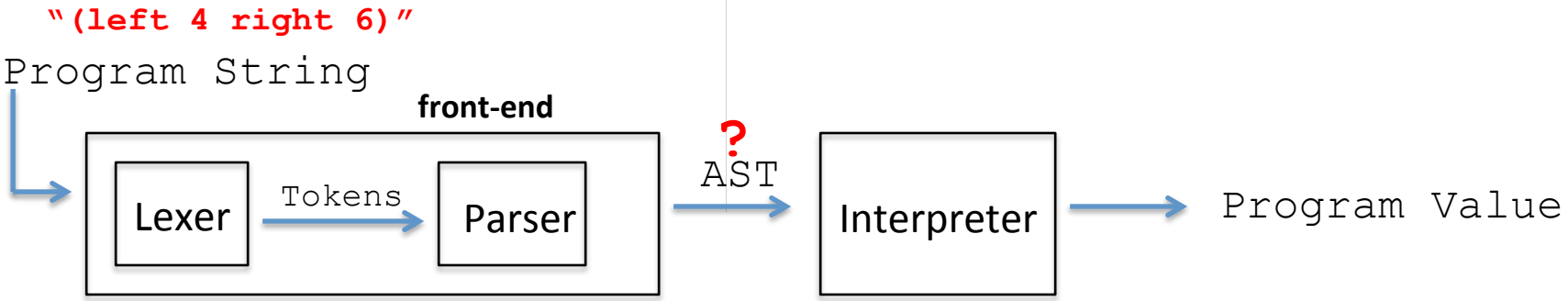

```
> (parser "(left 4 right 6)")
```

Structural Overview Of P/L Processing Systems



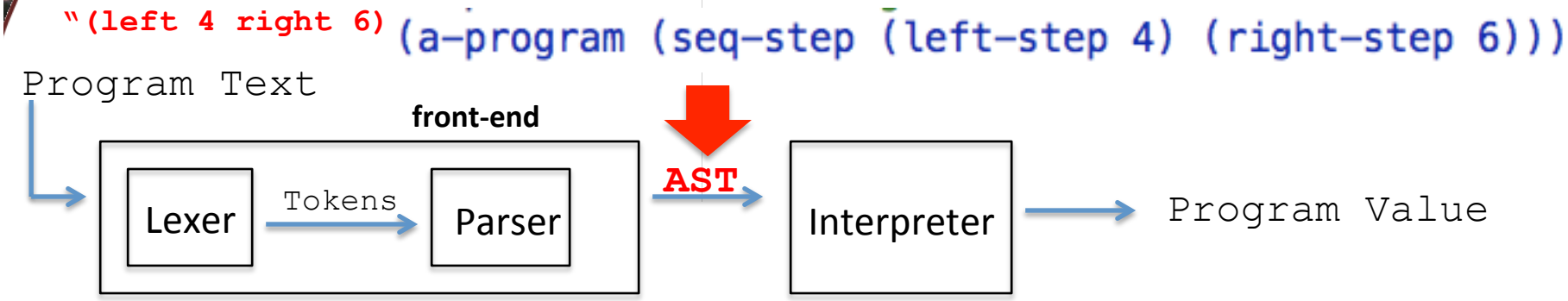

```
> (parser "(left 4 right 6)")
(a-program (seq-step (left-step 4) (right-step 6)))
```

Structural Overview Of P/L Processing Systems

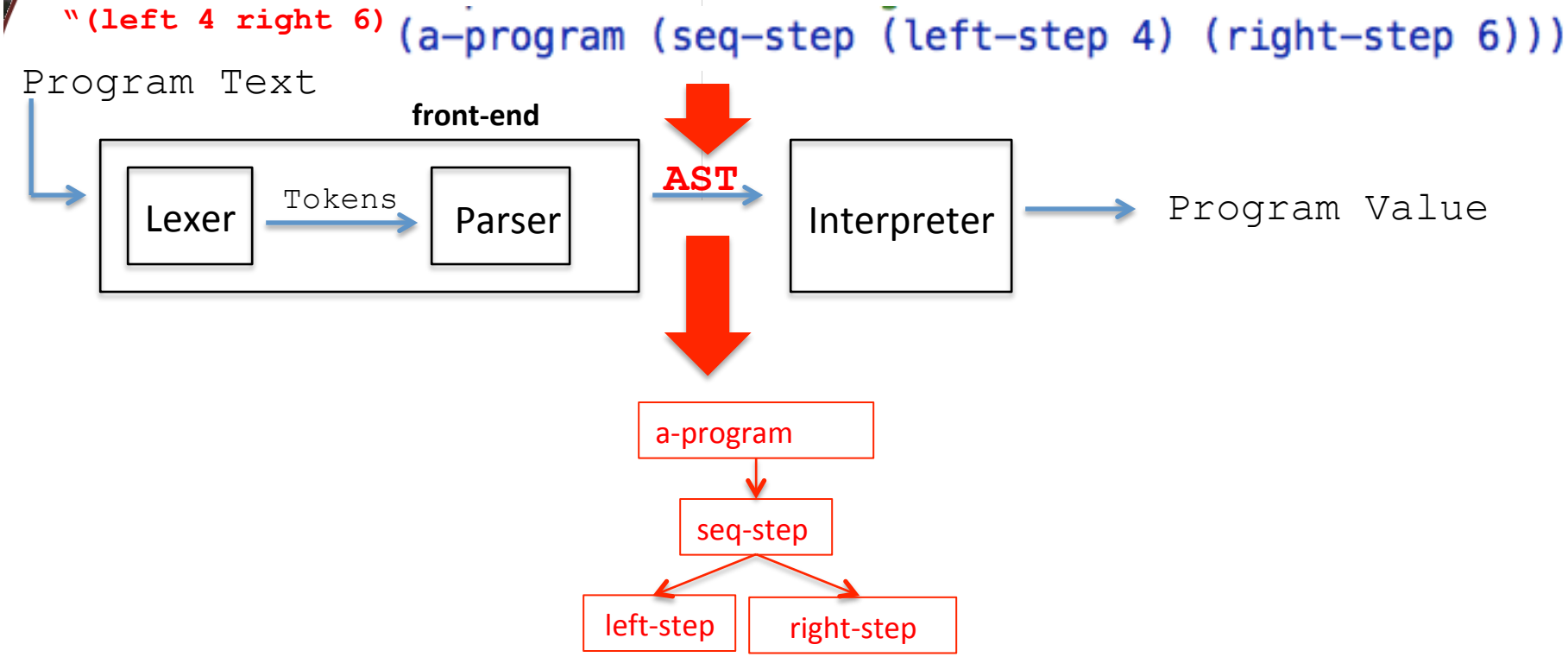



```
> (parser "(left 4 right 6)")
(a-program (seq-step (left-step 4) (right-step 6)))
```

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)`

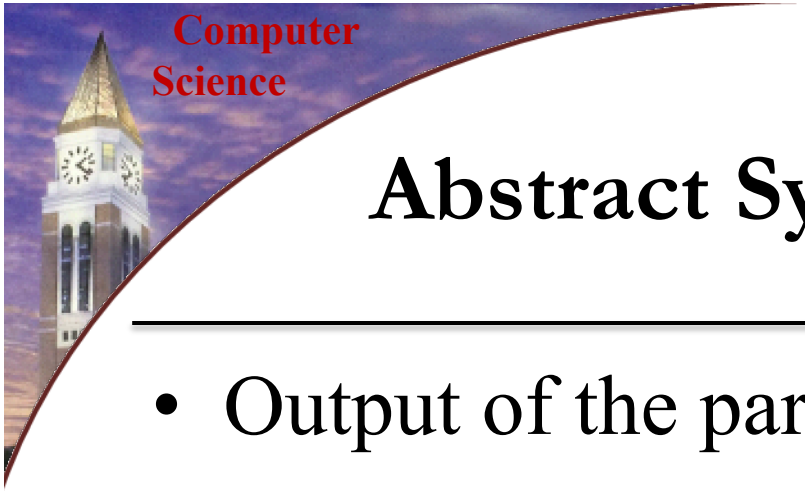
Expression ::= *Identifier*
`var-exp (var)`

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

What does a parser return to us?

Concrete Syntax

Abstract Syntax



Abstract Syntax: An Example

- Output of the parser: an AST

Concrete syntax for if-expression

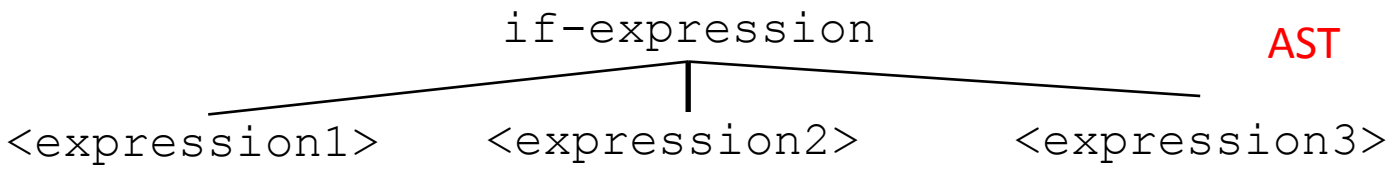
`if` <expression> `then` <expression> `else` <expression>

The Abstract Syntax

Concrete syntax for if-expression

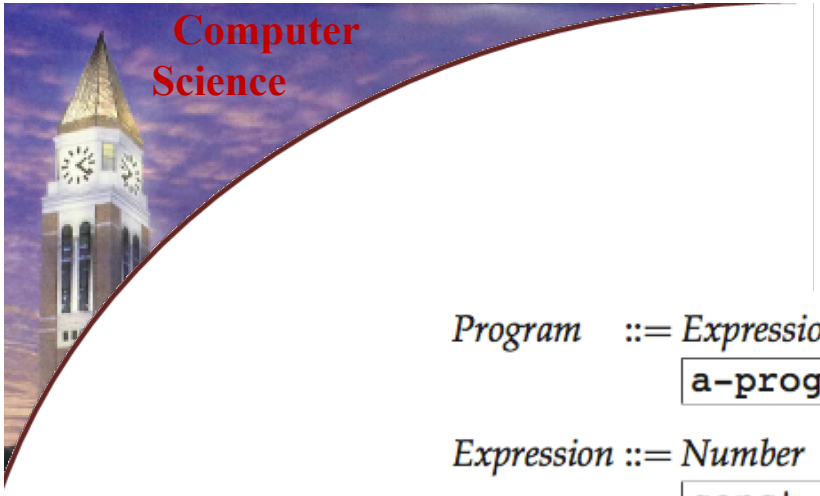
`if` <expression1> `then` <expression2> `else` <expression3>

The essential structure of `if-expression`



linearize AST

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



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

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

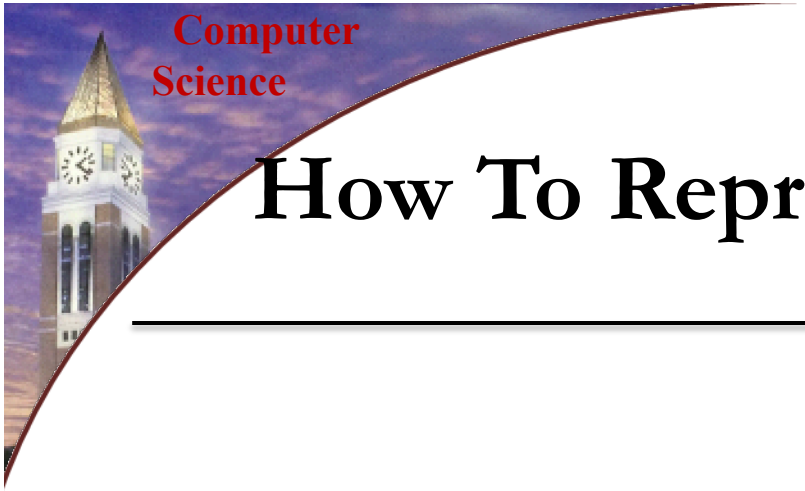
Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

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)`



How To Represent Abstract Syntax

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

What can you imagine from the above from?

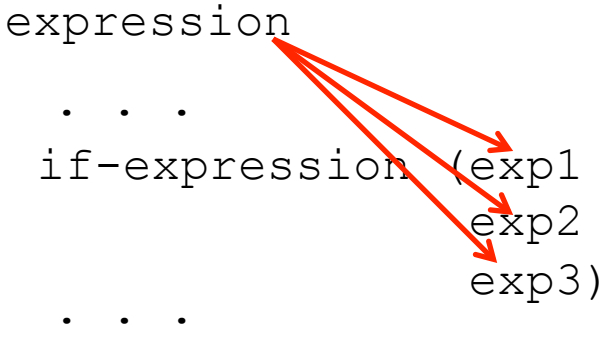
The Abstract Syntax

```
expression
```

exp1, exp2, exp3 are all expressions

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

  . . .
```

The Abstract Syntax



The Abstract Syntax

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

The Abstract Syntax

```
expression

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

The Abstract Syntax

```
(expression

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

The Abstract Syntax

```
(define-datatype expression
   . . .
  if-expression ( (exp1 expression?)
                  (exp2 expression?)
                  (exp3 expression?)  )
   . . .
)
```

The Abstract Syntax

```
(define-datatype expression

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

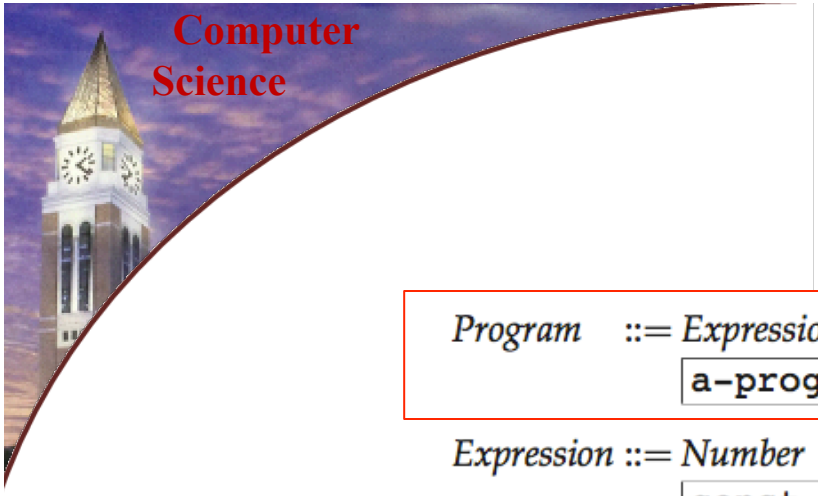
The Abstract Syntax

```
(define-datatype expression expression?
   . . .
  (if-expression ( (exp1 expression?)
                   (exp2 expression?)
                   (exp3 expression?)  ) )
   . . .
)
```

The Abstract Syntax

```
(define-datatype expression expression?
   . . .
  (if-expression ( (exp1 expression?)
                   (exp2 expression?)
                   (exp3 expression?)  ) )
   . . .
)
```

Abstract Syntax for
<expression>



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

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

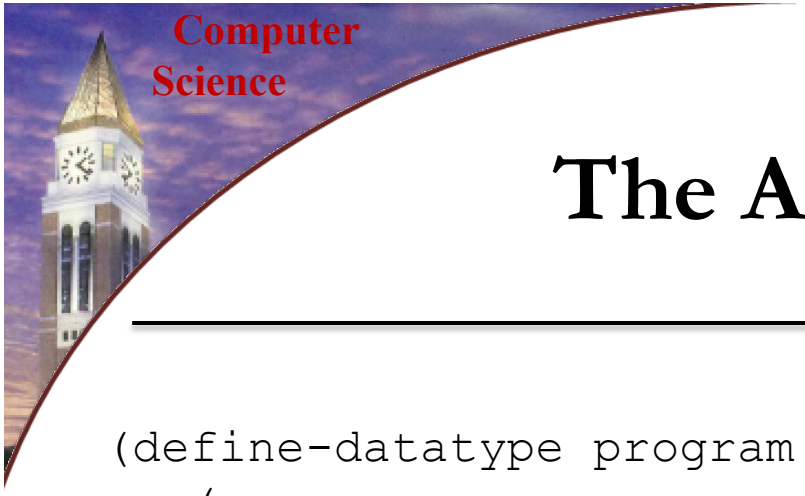
Expression ::= `-(` *Expression* `,` *Expression* `)`
`diff-exp (exp1 exp2)`

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)`



The Abstract Syntax

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

(define-datatype expression expression?

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

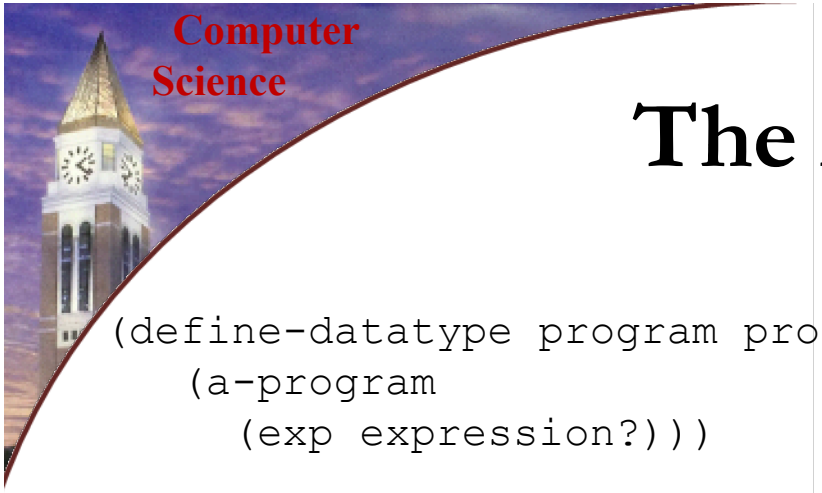
Abstract Syntax for
<expression>

The Abstract Syntax

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

(define-datatype expression expression?

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

Abstract Syntax for **<program>**

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

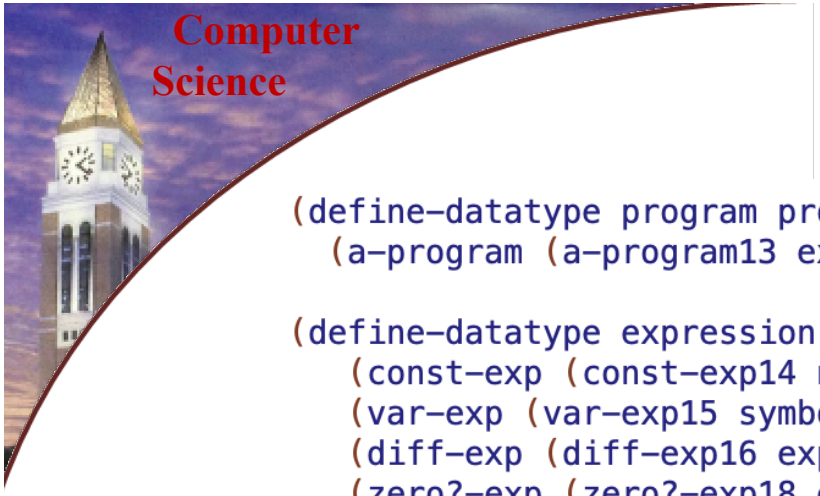


The Abstract Syntax

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

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

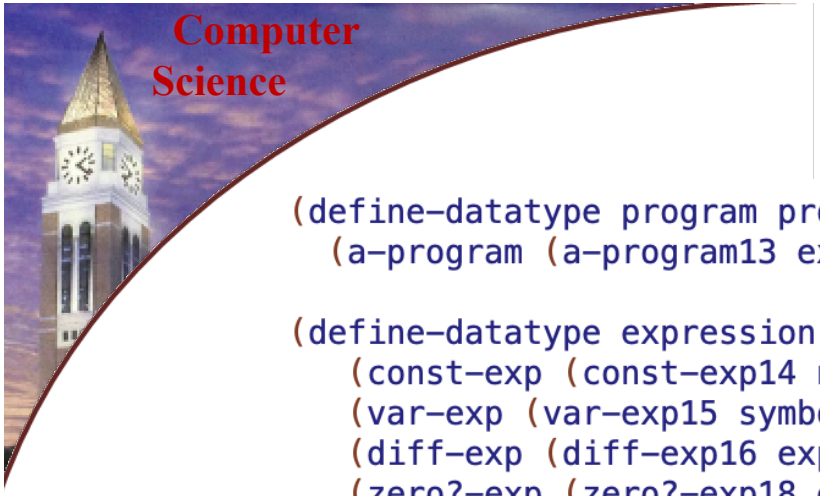
Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*

`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)))
```

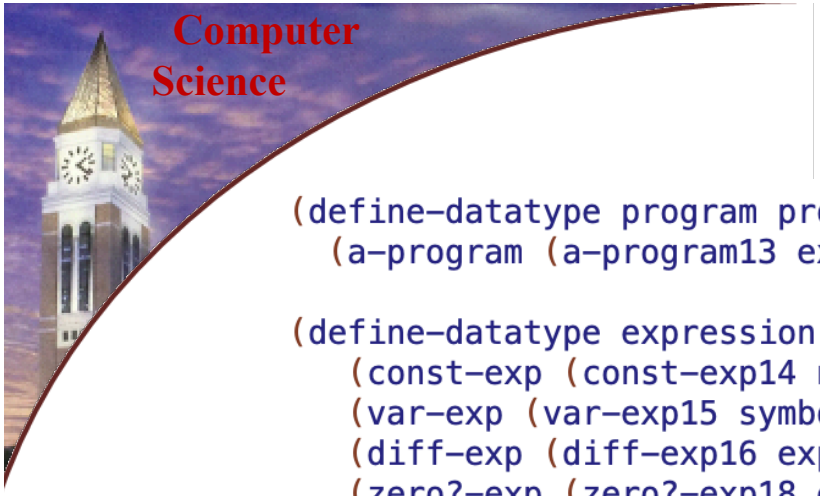


abstract syntax

```
(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?)))
```

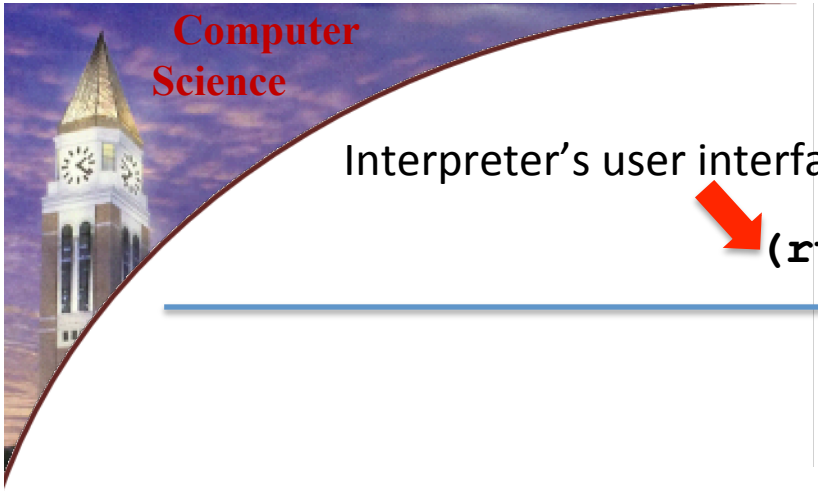
```
(define lexical-spec
  '(
    (whitespace (whitespace) skip)
    (comment ("#" (arbno (not #\newline))) skip)
    (num (digit (arbno digit)) number)
    (identifier (letter (arbno (or letter digit "_" "-" "?"))) symbol)))


(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)))
```

(show-data-types)

concrete syntax

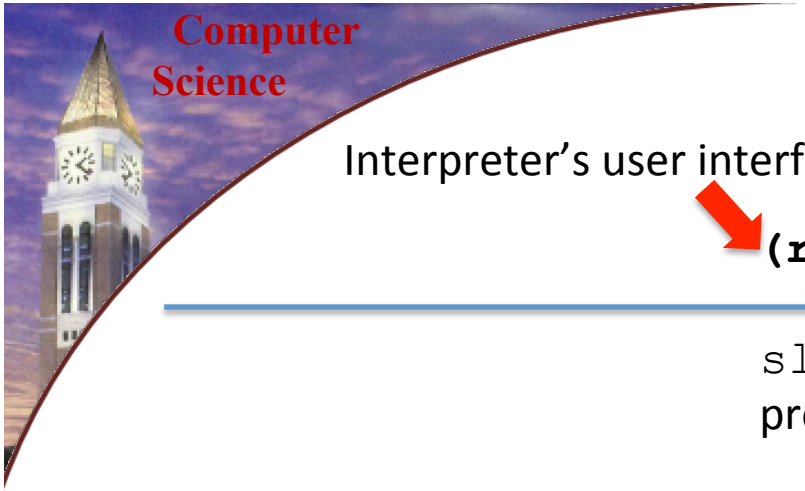
HW06

```
<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"
```



Interpreter's user interface

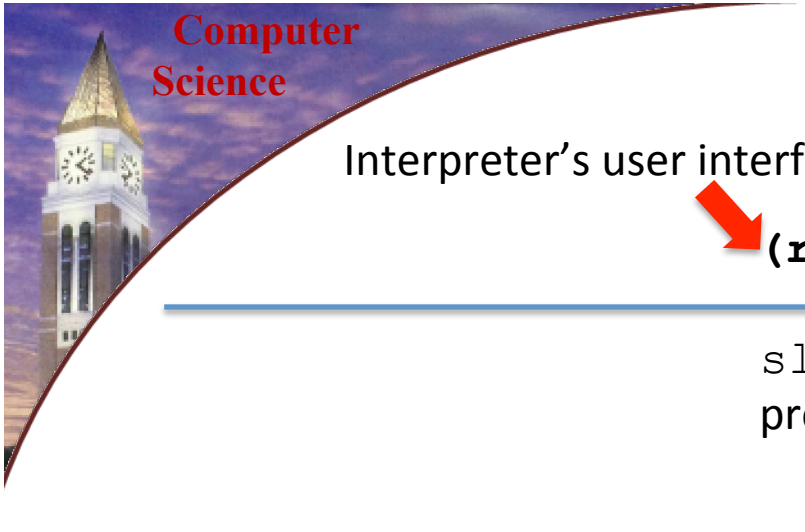
```
(run program-string)
```



Interpreter's user interface

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

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

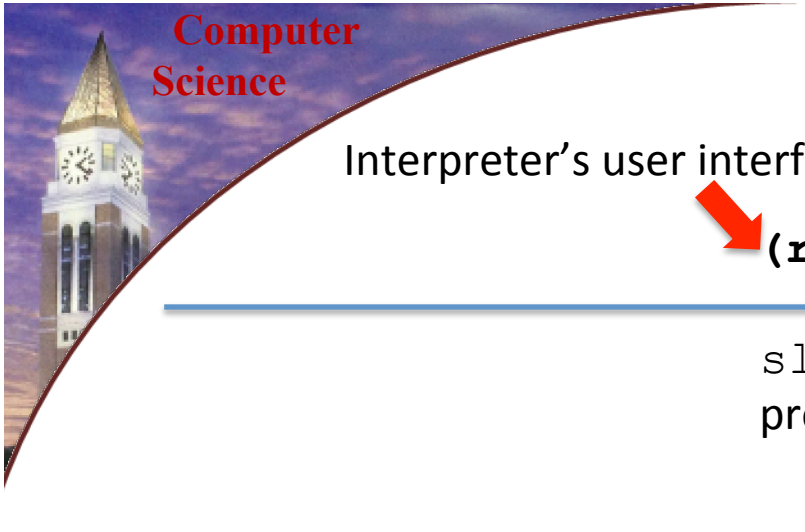


Interpreter's user interface

`(run program-string)`

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

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



Interpreter's user interface

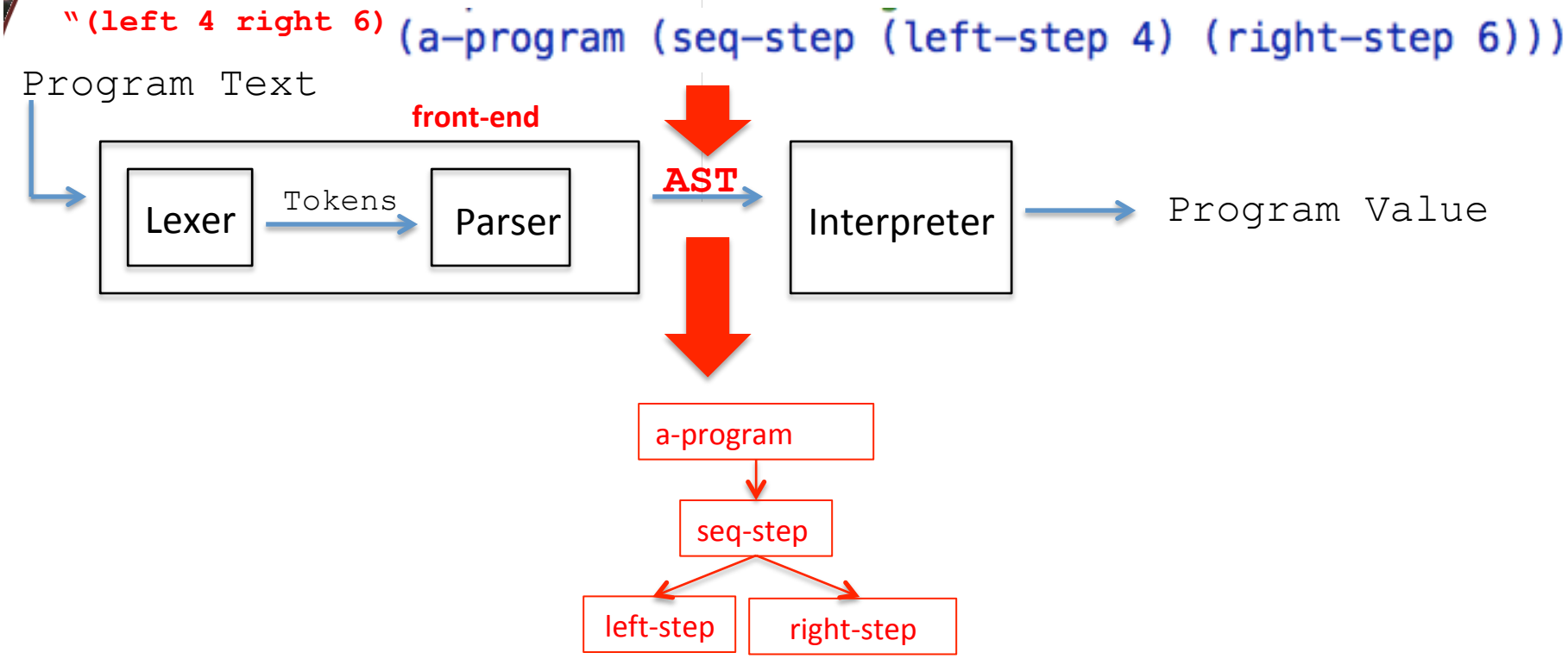
`(run program-string)`

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

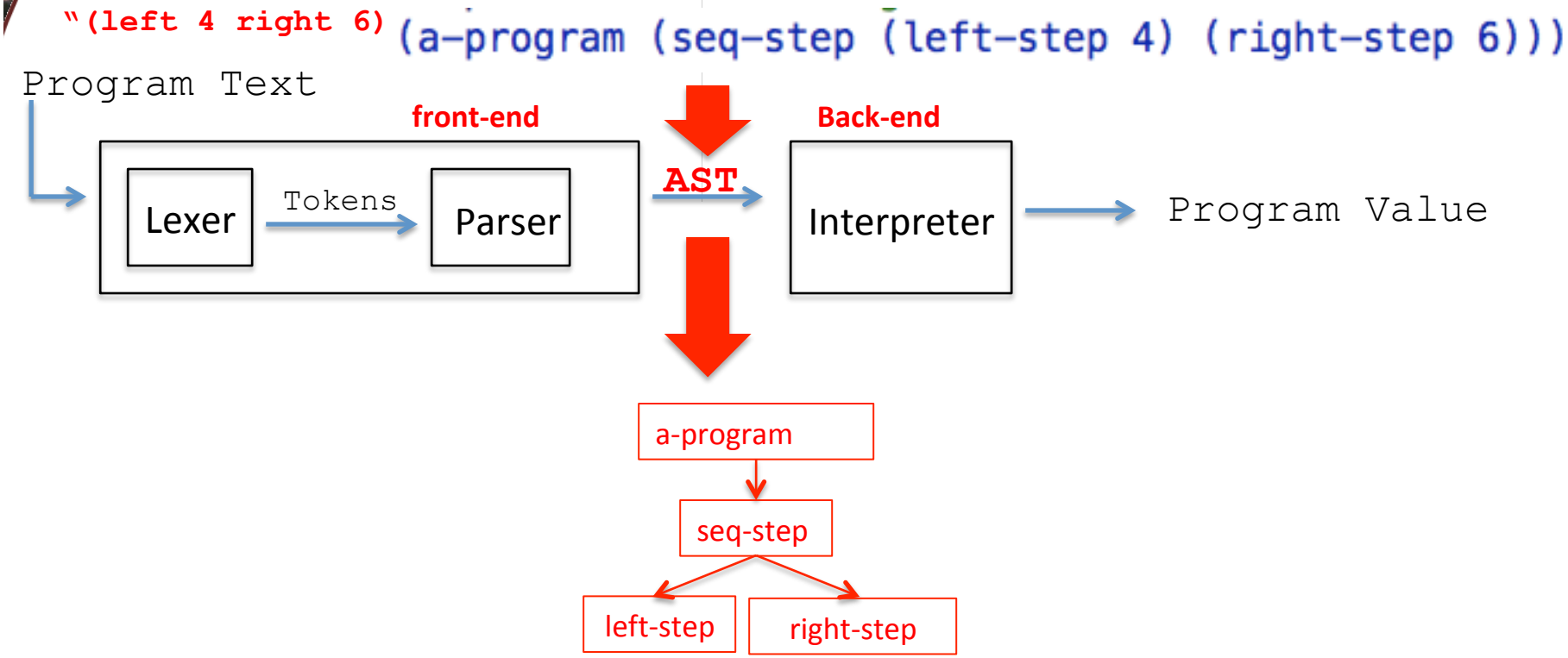
`ast` = (`parser` programing-string)

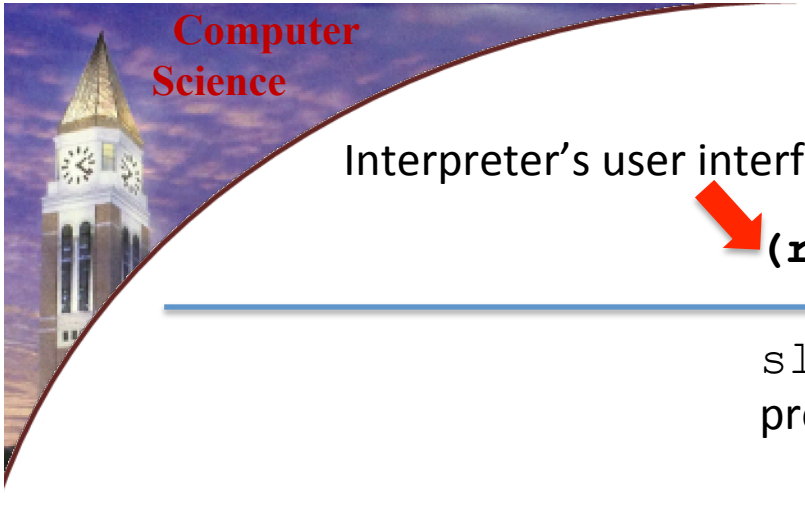
Back-end

Internal Representation of Program Values



Internal Representation of Program Values





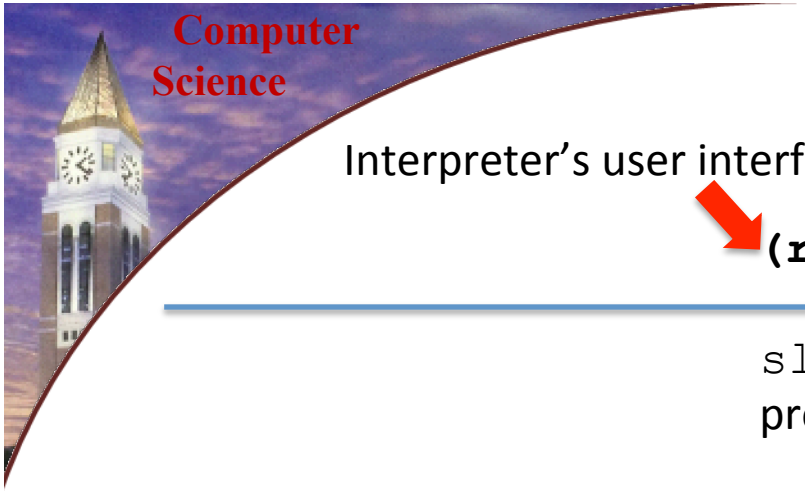
Interpreter's user interface

`(run program-string)`

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

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

Back-end



Interpreter's user interface

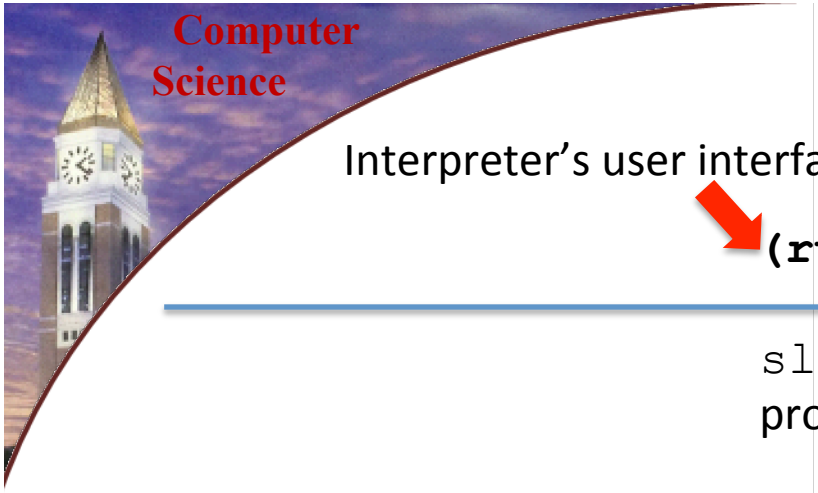
`(run program-string)`

`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)

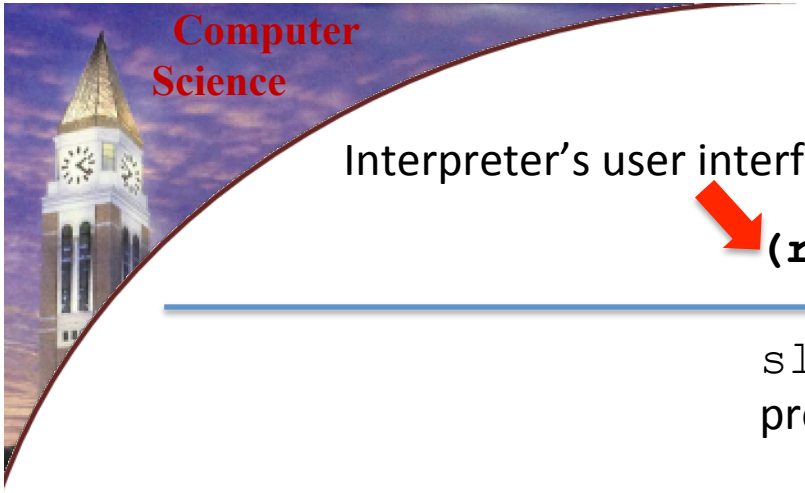
(value-of `ast`)

- (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`)
- . . .

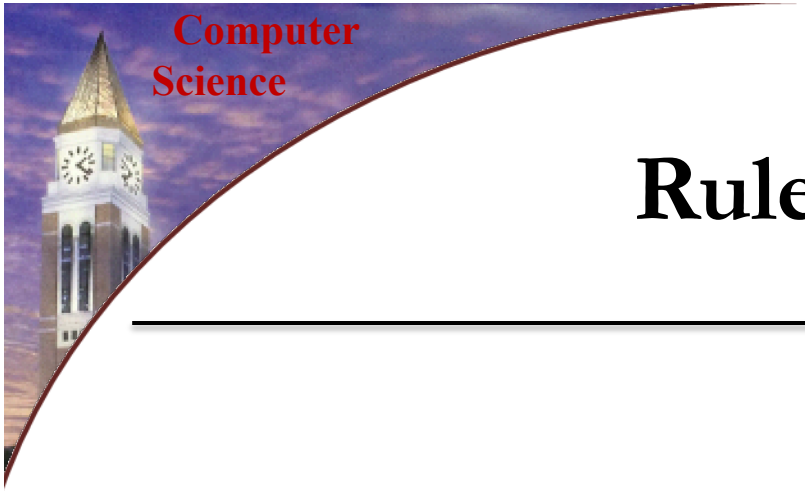
Back-end

(value-of-program `ast`)

(value-of-expr `ast`)

define-datatype:

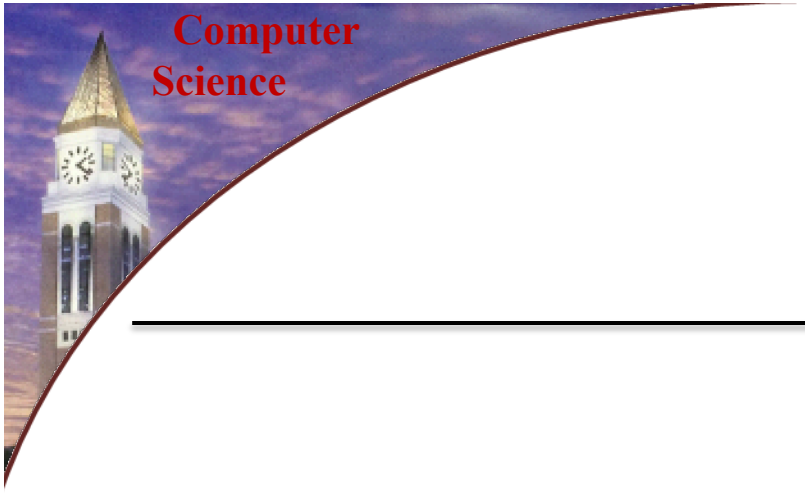
- think of each non-terminal in a grammar production as one data type
- each non-terminal takes different variants (num-expr, str-expr, ... etc.)



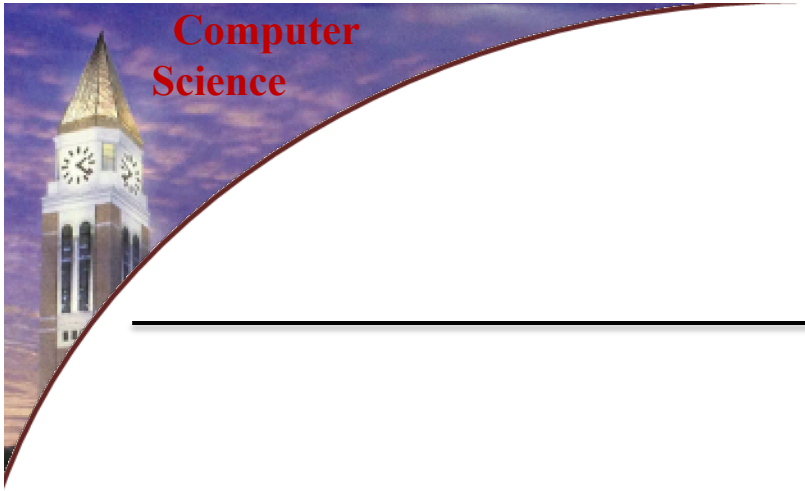
Rule Of Thumb

Every expression will return a value!

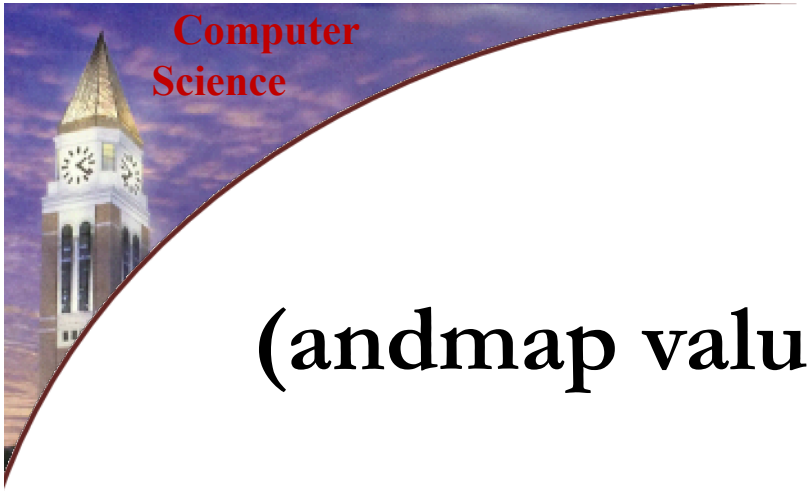
For a program consisting of multiple expressions, the **last** expression's value will be the value of the overall program!



`map` is not enough!

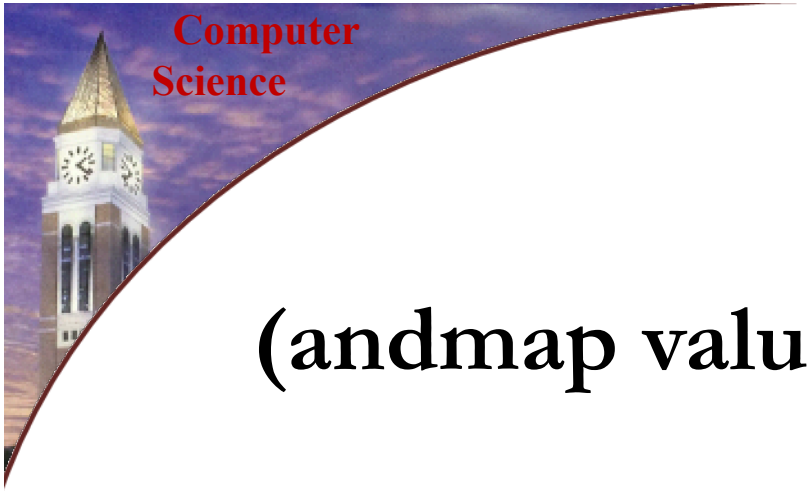


andmap !



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

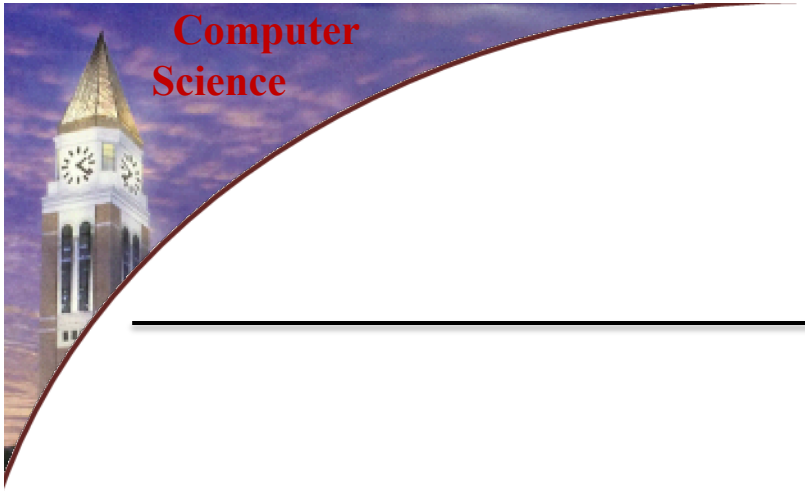
andmap will apply **value-of-expr** to all the expressions in **list-of-expr**, and return the value of **the last expression** in the list ! Which is exactly what we need to evaluate a program which may consists of multiple expressions!



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!



HW06

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

ast = (parser a-program-string)