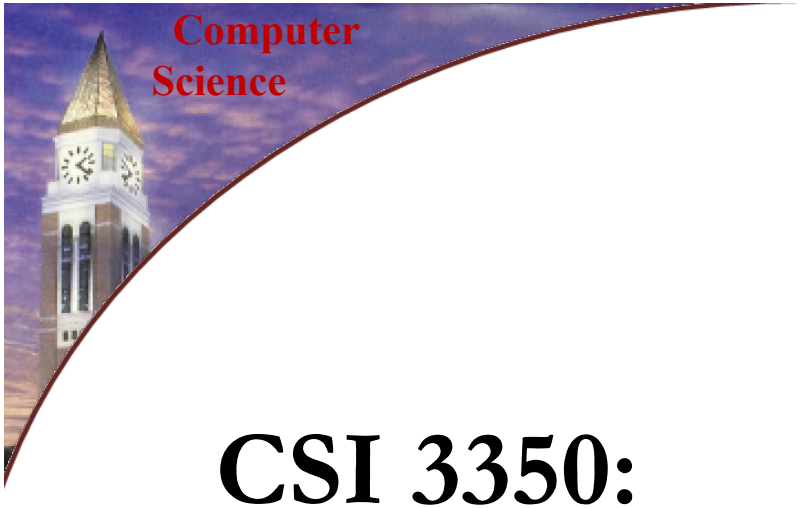


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University

Road Ahead -

HW05

↓

HW06

↓

Exam02

↓

HW07

↓

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

Data Type Definition (ADT)

- For complicated types, manual definition tedious
- Plant errors in data type definitions

an observation

- It has recipe!
  - Constructors, Predicates, Extractors

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

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

define-datatype

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

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)}*)}+

(on slide 6)

What Do We Get?

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

```
> (Env? #f)
#f
> (Env? (empty-env))
#t
> (Env? (extend-env 'x 20 (empty-env)))
#t
```

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

Summary on `define-datatype`

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

- Each variant has a variant-name with 0 or more fields
- Each field has its own name and associated predicate
- A new constructor for each variant is created
- Type predicate name is bound to a predicate, which determines if its argument is a value of the type

- **No two types may have the same name**
- **No two variants have the same name**
- **Type names may not be used as variant names**
- **Each field predicate must be a Scheme predicate**

} Constraints

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
- …

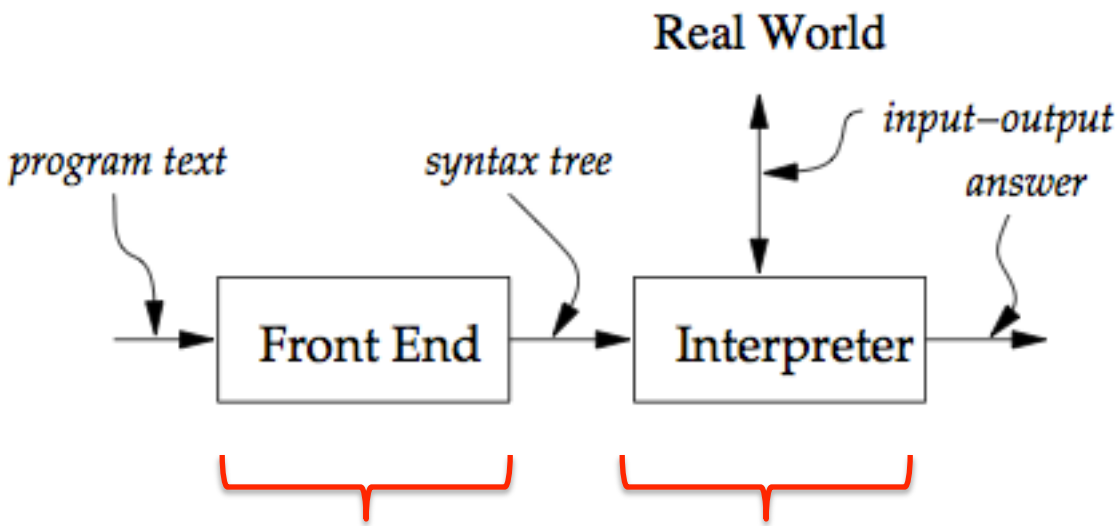
Our Learning Style

- 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
```

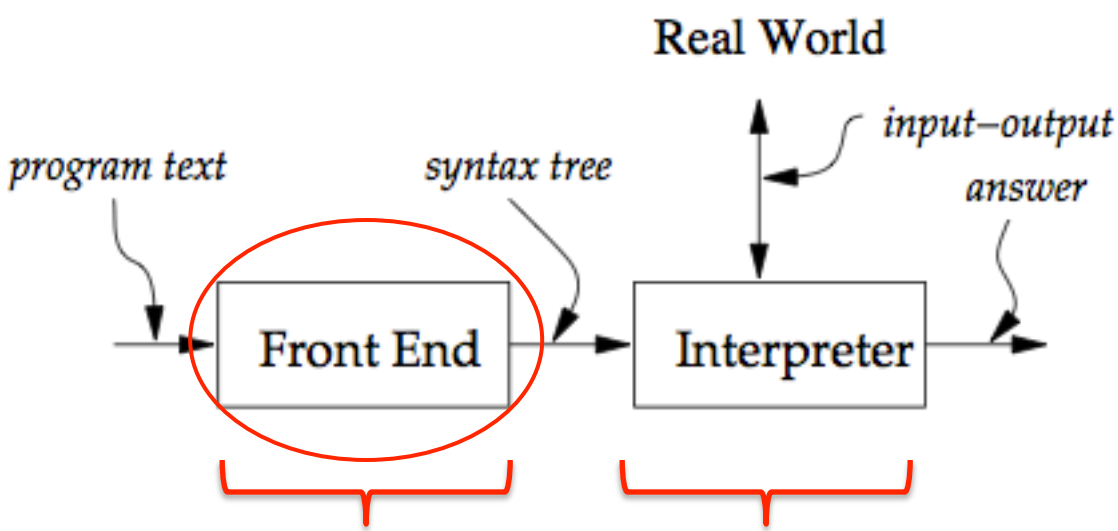


scanning & parsing based on lexical and grammar spec

majority of your work here!

The Basic Form Of The Interpreter

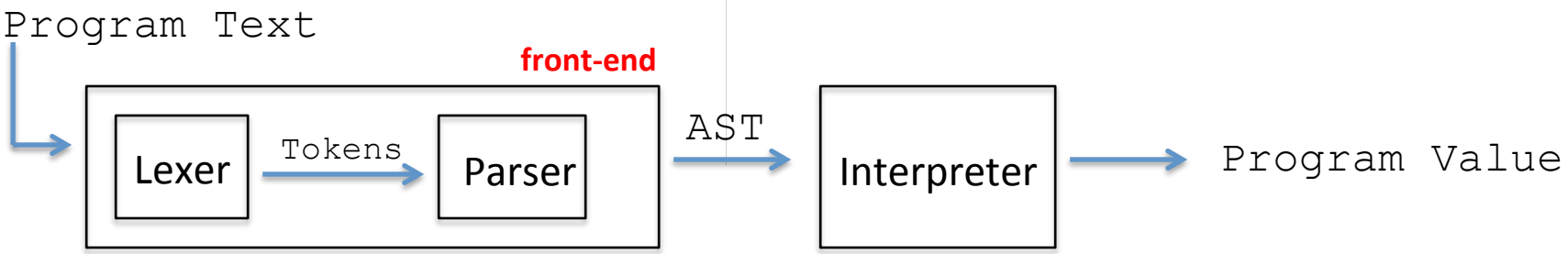
```
(value-of exp env) = val
```



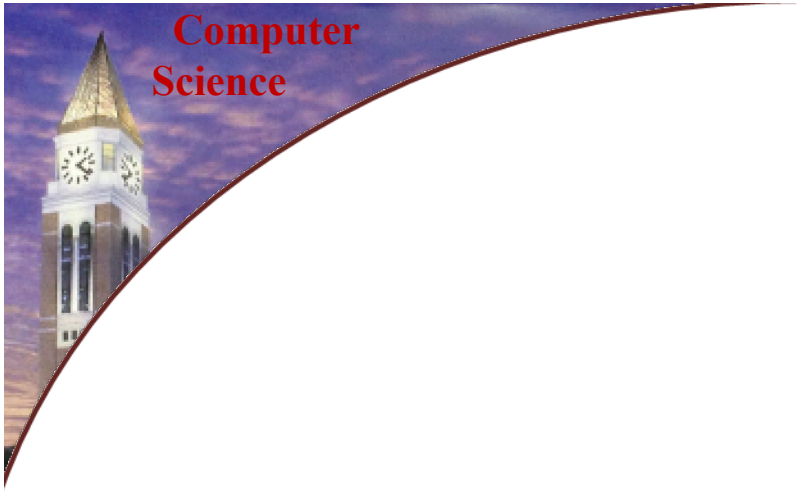
scanning & parsing based on lexical and grammar spec

majority of your work here!

Structural Overview



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

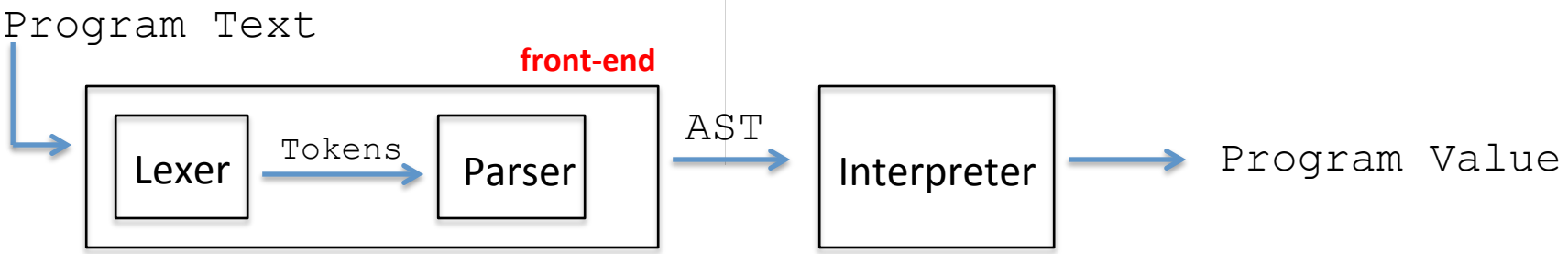


IMPLEMENTING A PROGRAMMING LANGUAGE OF YOUR DESIGN

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

token
actions

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

token are used in your grammar

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

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 `AST` datatype with `define-datatype` according to the lexical and grammar specifications.

SLLGEN Boiler Plate

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

Use this function to display the define-datatype expression used to generate the AST. Take some time to read it...

SLLGEN Boiler Plate

`parser` is a one argument function that takes a string, scans & parses it and generates an 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 34** to find it out -