Why do we need an interpreter?
- Abstractions let us bury details and focus on use of modules to solve large systems
- Need to unwind abstractions at execution time to deduce meaning
- Have seen such a process – Environment Model
- Now want to describe that process as a procedure

Role of each part of the interpreter
- **Lexical analyzer**
  - break up input string into "words" called tokens
- **Parser**
  - convert linear sequence of tokens to a tree
  - like diagramming sentences in elementary school
  - also convert self-evaluating tokens to their internal values
  - #f is converted to the internal false value
- **Evaluator**
  - follow language rules to convert parse tree to a value
  - read and modify the environment as needed
- **Printer**
  - convert value to human-readable output string

Goal of lecture
- Implement an interpreter
- Only write evaluator and environment
  - use scheme's reader for lexical analysis and parsing
  - use scheme's printer for output
  - to do this, our language must look like scheme
- Call the language scheme*
  - All names end with a star
- Start with interpreter for simple arithmetic expressions
- Progressively add more features

1. Arithmetic calculator
Want to evaluate arithmetic expressions of two arguments, like:

```
(plus* 24 (plus* 5 6))
```
1. Arithmetic calculator

(define (tag-check e sym) (and (pair? e) (eq? (car e) sym)))
(define (sum? e) (tag-check e 'plus*))
(define (eval exp)
  (cond
   ((number? exp) exp)
   ((sum? exp)    (eval-sum exp))
   (else
    (error "unknown expression " exp))))
(define (eval-sum exp)
  (+ (eval (cadr exp)) (eval (caddr exp))))
(eval '(plus* 24 (plus* 5 6)))

1. Things to observe

• cond determines the expression type
• no work to do on numbers
  • scheme's reader has already done the work
  • it converts a sequence of characters like "24" to an internal binary representation of the number 24
• eval-sum recursively calls eval on both argument expressions

1. More Complex Expressions

(define x* (plus* 5 6))
(define y* (plus 24 x*))
(define z* (plus 43 y*))

We are just walking through a tree ...

1. We are just walking through a tree ...

(eval-sum (plus* 24 (plus* 5 6)))

sum? checks the tag

1. We are just walking through a tree ...

We are just walking through a tree ...

(plus* 24 (plus* 5 6))

• What are the argument and return values of eval each time it is called in the evaluation of line 17?

(eval 24)
(eval 5)
(eval 6)
(eval '(plus* 24 (plus* 5 6)))
(eval-sum '(plus* 5 6))
(eval-sum '(plus* 24 (plus* 5 6)))
(eval '(plus* 24 (plus* 5 6)))

More Complex Expressions

(plus* 24 (plus* 5 6))
(plus* (plus* 43 (plus* 24 (plus* 5 6))) (plus* 43 (plus* 24 (plus* 5 6))))
(plus* (plus* 43 (plus* 24 (plus* 5 6))) (plus* 43 (plus* 24 (plus* 5 6))))

(define x* (plus* 5 6))
(define y* (plus 24 x*))
(define z* (plus 43 y*))
(plus z* z*)
2. Names

- Extend the calculator to store intermediate results as named values
  
  ```scheme
  (define* x* (plus* 4 5))  ; store result as x*
  (plus* x* 2)              ; use that result
  ```

- Store bindings between names and values in a table

- What are the argument and return values of eval each time it is called in lines 36 and 37?

- Show the environment each time it changes during evaluation of these two lines.

```scheme
(define (eval exp)
  (cond
    ((number? exp) exp)
    ((sum? exp)    (eval-sum exp))
    ((symbol? exp) (lookup exp))
    ((define? exp) (eval-define exp))
    (else
     (error "unknown expression " exp))))
```

```scheme
(define environment (make-table))
```

```scheme
(define (lookup name)
  (let ((binding (table-get environment name)))
    (if (null? binding)
        (error "unbound variable: " name)
        (binding-value binding)))))
```

```scheme
(define (eval-define exp)
  (let ((name          (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! environment name (eval defined-to-be))
    'undefined))
```

2. Names ...

```scheme
(eval '(define* x* (plus* 4 5)))
```

```scheme
(eval '(plus* x* 2))
```

2. Things to observe

- Use scheme function `symbol?` to check for a name
  - the reader converts sequences of characters like "x*" to symbols in the parse tree

- Can use any implementation of the table ADT

- `eval-define` recursively calls `eval` on the second subtree but not on the first one

- `eval-define` returns a special undefined value

3. Conditionals and if

- Extend the calculator to handle predicates and if:
  
  ```scheme
  (if* (greater* y* 6) (plus* y* 2) 15)
  ```

- `greater*` an operation that returns a boolean
- `if*` an operation that evaluates the first subexp, checks if value is true or false

- What are the argument and return values of eval each time it is called in line 32 (above)?

2. Evaluation of page 2 lines 36 and 37

```scheme
(eval '(define* x* (plus* 4 5)))
```

```scheme
(eval '(plus* x* 2))
```

```scheme
(eval 4) ==> 4
(eval 5) ==> 5
=> 9
=> 11
```

```scheme
(eval 2) == 2
```

```scheme
(eval '(define* x* (plus* 4 5)))
```

```scheme
(eval 'x*) ==> 9
```

```scheme
(names values
      x* 9
)
We are just walking through a tree ...

Then (eval (greater* y* 6)) or (eval 15)

We evaluate the expression:

\[
\text{if* (greater* y* 6) (plus* y* 2) 15}
\]

We evaluate the procedure (eval 'greater* y* 6)

\[
\text{if* (greater* y* 6) (plus* y* 2) 15}
\]

\[
\text{if* (greater* y* 6) (plus* y* 2) 15}
\]

Note: if* is stricter than Scheme's if

3. Conditionals and If

3. Things to observe

4. Store operators in the environment

4. Store operators in the environment
Environment after eval 4 line 36

<table>
<thead>
<tr>
<th>names</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>z*</td>
<td>9</td>
</tr>
<tr>
<td>true*</td>
<td>#t</td>
</tr>
<tr>
<td>greater*</td>
<td></td>
</tr>
<tr>
<td>plus*</td>
<td></td>
</tr>
</tbody>
</table>

Evaluation of eval 4 line 37

```scheme
(eval '(define* z* 9))
(eval '(plus* 9 6))
(eval '(if* true* 10 15))
```

4. Things to observe

- applications must be last case in eval
- no tag check
- apply is never called in line 38
- applications evaluate all subexpressions
- expressions that need special handling, like `if*`, gets their own case in eval

5. Environment as explicit parameter

- change from
  ```scheme
  (eval '(plus* 6 4))
  ```
  to
  ```scheme
  (eval '(plus* 6 4) environment)
  ```
- all procedures that call `eval` have extra argument
- `lookup` and `define` use environment from argument
- No other change from evaluator 4
- Only nontrivial code: case for `application?` in `eval`
6. Defining new procedures

- Want to add new procedures
- For example, a `scheme*` procedure:

  ```scheme
  (define* twice* (lambda* (x*) (plus* x* x*))
  (twice* 4))
  ```

- Strategy:
  - Add a case for `lambda*` to `eval`
  - Extend `apply` to handle compound procedures
  - Implement environment model

---

Implementation of `lambda*`

- `eval '(lambda* (x*) (plus* x* x*)) GE`
- `eval-lambda '(lambda* (x*) (plus* x* x*)) GE`
- `make-compound '(x*) '(plus* x* x*) GE`
- `list 'compound '(x*) '(plus* x* x*) GE`

- `symbol compound`
- `symbol plus*`
- `symbol x*`

---

Defining a named procedure

- `eval '(define* twice* (lambda* (x*) (plus* x* x*)) GE)`

```
(names values
  x* 9
  plus* #t
  twice*)
```

- `scheme procedure +`

---

Implementation of `apply` (1)

- `eval '(twice* 4) GE`
- `apply (eval 'twice* GE)`
- `map (lambda (e) (eval e GE)) '(4))`
- `apply (list 'compound '(x*) '(plus* x* x*)) GE`
- `eval '(plus* x* x*)`
- `extend-env-with-new-frame '(x*) '(4) GE`
- `eval '(plus* x* x*) E1`

- `A` name value `x*` 4

---

Implementation of `apply` (2)

- `eval '(plus* x* x*) E1`
- `apply (eval 'plus* E1)`
- `map (lambda (e) (eval e E1)) '(x* x*))`
- `apply '(primitive #[add]) (list (eval 'x* E1) (eval 'x* E1))`
- `apply '(primitive #[add]) '(4 4)`
- `scheme-apply #[add] '(4 4)`
- `8`
Implementation of environment model

- Environment = list<table>

Environment model code (part of eval 6)

```scheme
(define (extend-env-with-new-frame names values env)
  (let ((new-frame (make-table)))
    (make-bindings! names values new-frame)
    (cons new-frame env)))

(define (make-bindings! names values table)
  (for-each
   (lambda (name value) (table-put! table name value))
   names values))

; the initial global environment
(define GE
  (extend-env-with-new-frame
   (list 'plus* 'greater*)
   (list (make-primitive +) (make-primitive >))
   nil))

; lookup searches the list of frames for the first match
(define (lookup name env)
  (if (null? env)
      (error "unbound variable: " name)
      (let ((binding (table-get (car env) name)))
        (if (null? binding)
            (lookup name (cdr env))
            (binding-value binding)))))

; defines changes the first frame in the environment
(define (eval-define exp env)
  (let ((name (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! (car env) name (eval defined-to-be env))
    'undefined))

eval '(define* twice* (lambda* (x*) (plus* x* x*)) GE)
eval '(twice* 4) GE)
```

Summary

- Cycle between eval and apply is the core of the evaluator
  - eval calls apply with operator and argument values
  - apply calls eval with expression and environment
  - no pending operations on either call
    - an iterative algorithm if the expression is iterative
- What is still missing from scheme*?
  - ability to evaluate a sequence of expressions
  - data types other than numbers and booleans

Everything in these lectures would still work if you deleted the stars from the names!