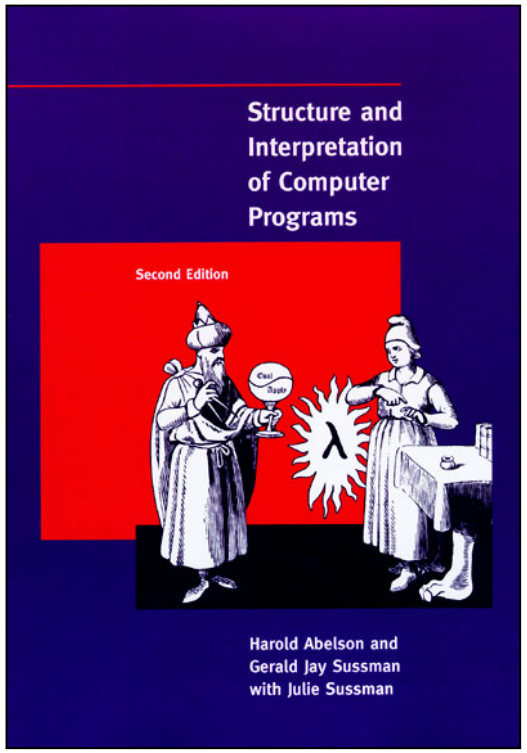
Lecture 2 — August 5, 2015

- **Today:**
  - EBNF
  - Scheme
- **Readings:**
  - *Structure and Interpretation of Computer Programs:* Section 1.3
  - [Finish reading *SICP* 1.1, 1.2 if you haven't already]
- **Exercises, due Monday, August 10:**
  - Do 1.2, 1.3, 1.11, 1.12, 1.16, putting all solutions in a file named `hw1.scm`
  - Turn-in instructions to follow



CSE130 Summer Session II

EBNF: Java `import` statement example

- Extended Backus-Naur form notation (EBNF):
  - [*x*] means zero or one occurrences of *x*
  - { *x* } means zero or more occurrences of *x*

ImportDeclaration → **`import`** *JavaIdentifier* {**`.`** *JavaIdentifier*} [**`. *`**] **`;`**

- Concise, easier to read, less need for ε
- Compare to non EBNF version:

ImportDeclaration → **`import`** *JavaIdentifier DottedNames OptStar* **`;`**

DottedNames → ε | *DottedNames* **`.`** *JavaIdentifier*

OptStar → ε | **`. *`**

And Now Scheme

- ***Scheme*** is a variant of LISP, invented at MIT by Guy Steele and Gerald Sussman
- Inspired by making LISP even more like the 1930s mathematical logic on which it was based: ***Lambda Calculus***
- Simple syntax, but very powerful language
- Scheme Rules:
  - Legal expressions have rules for construction from simpler pieces
  - (Almost) Every expression has a value, which is "returned" when an expression is evaluated
  - Every value has a type

Kinds of Language Constructs

- Primitives
- Means of Combination
- Means of Abstraction

Language Elements: Primitives

- So-called ***self-evaluating primitives:*** the value of the expression is just the object itself:
  - Numbers: `29`, `-35`, `1.34`, `1.2e5`
  - Strings: `"this is a string"`
  - Characters: `#\a`, `#\b`, `#\z`, `#\newline`
  - Booleans: `#t`, `#f`

- Built-in ***procedures*** to manipulate primitives:
  - For numbers: `+`, `-`, `*`, `/`, `>`, `<`, `>=`, `<=`, `=`, `abs`, `max`
  - For strings: `string-length`, `string=?`, `substring`, `string-append`
  - For characters: `char=?`, `char<?`, `char-numeric?`, `char-upcase`
  - For booleans: `and`, `or`, `not`

Language Elements: Primitives

- Built-in procedure names:
  - `+`, `-`, `*`, `/`, `>`, `<`, `>=`, `<=`, `=`, `abs`, `max`, `string-length`, `string=?`, `substring`, `string-append`, `char=?`, `char<?`, `char-numeric?`, `char-upcase`, `and`, `or`, `not`
- These names are expressions, which means they have values
- So, what would be the value of '`+`'?
- Syntax:
  - A valid Scheme identifier
- Semantics:
  - These are *self-evaluating,* so no further evaluation steps are needed

Scheme Identifier Syntax

- Alphabet: ASCII characters
- Case is not important, e.g., `FUN`, `fun`, `Fun`
  - case matters in variants, like Racket
- Scheme names are flexible: `pi*2`, `<object>`, `*example*`, `<?`

Identifier → *Initial Subsequents* | `+` | `-` | `...`

Subsequents → *Subsequent Subsequents* | ε

Initial → *Letter* | `!` | `$` | `%` | `&` | `*` | `/` | `:` | `<` | `=` | `>` | `?` | `~` | `_` | `^`

Subsequent → *Initial* | *Digit* | `.` | `+` | `-` | `@`

Letter → `a` | `b` | … | `z`

Digit → `0` | `1` | … | `9`

Language Elements: Combinations

- The ***apply operation:*** Apply a procedure to a series of arguments:

```
(+ 22 34)
```

(We can also say that we are *applying an operator to its operands.*)

- Syntax:
  - Open parenthesis at the start
  - Close parenthesis at the end
  - Expression (whose value is a procedure)
  - Other expressions, whose values become arguments
- Semantics:
  - Evaluate the sub-expressions, then apply the evaluated procedure to the evaluated arguments

Language Elements: Combinations

- Can use nested combinations-- Just apply rules recursively:

```
(+ (* 2 3) 4)
```

10

```
(* (+ 3 4) (- 8 2))
```

42

```
(+ (+ 3 (+ 15 4))
   (+ (* 11 10) 2))
```

134

Language Elements: Abstractions

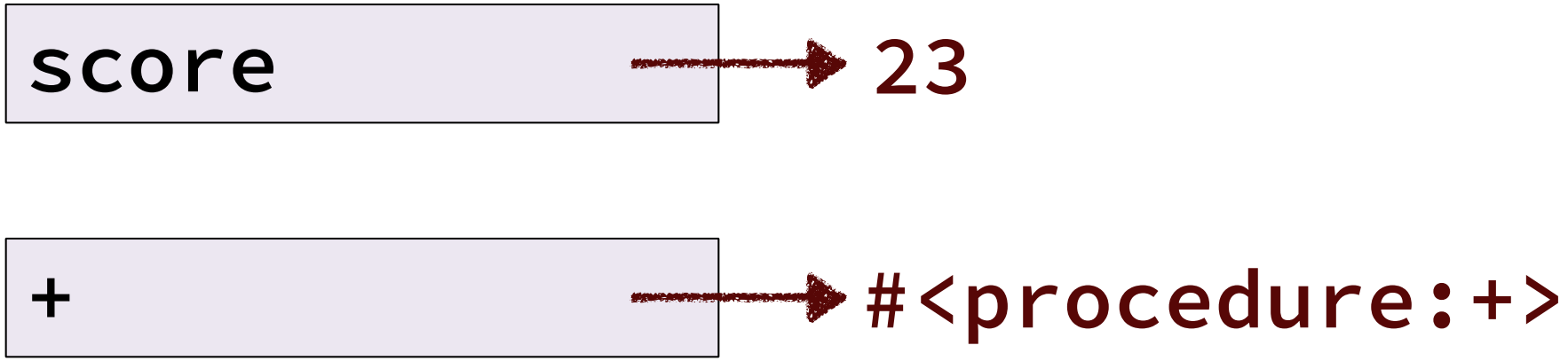
- So far, just a fancy calculator with prefix notation
- In order to ***abstract*** an expression, we need a way to give it a name. One way is with `define`:

```
(define score 23)
```

- This is a ***special form***
- Syntax:
  - It has the syntax of the apply operation, but it is not an application
  - `(` **`define`** *Name Expression* `)`
- Semantics:
  - Does not evaluate the second expression
  - But *does* evaluate the third expression
  - Rather, it pairs *Name* with value of the *Expression*

Language Elements: Abstractions

- To get the value of a name, just look up pairing in ***environment***:



- This is done for **+**, *****, **abs**, **max**…

```
(define total (+ 12 13))
(* 100 (/ score total))
```

→ 92

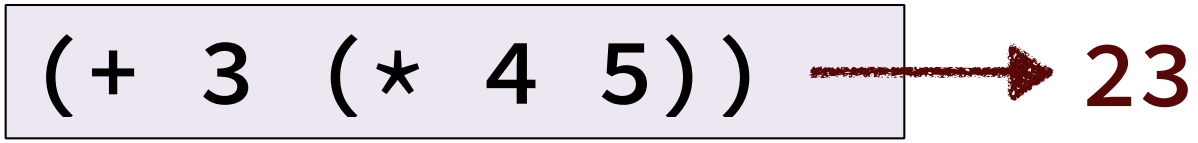
Scheme Basics

Rules for Evaluation

1. If ***self-evaluating***, return value.
2. If a ***name***, return value associated with name in environment
3. If a special form, …"do something special"
4. If a combination, then:
   - **Evaluate** all of the sub-expressions of combination (in any order)
   - **Apply** the operator to the values of the operands (arguments) and return result

Read-Eval-Print

- When an expression is entered into the computer, it is processed by a reader
- An internal representation is passed to an evaluator
  - application may involve more evaluations
- The resulting value is then printed




```
> +
#<procedure:+>
> (define total (+ 12 13))
(* 100 (/ score total))
92
> (+ 3 (* 4 5))
23
>
```

The `define` Special Form

Define-rule:

1. Evaluate second operand only
2. Name in first operand position is ***bound*** to that value
3. Overall value of the define expression is undefined

Operators are just names

```
(+ 3 5)
```
→ 8

```
(define fred +)
```
→ *undefined*

```
(fred 4 6)
```
→ 10

- What just happened?

Language Elements: Abstractions

- Procedures capture ways of doing things
- Special expression in Scheme—the ***lambda expression:***

```
(lambda (x) (* x x))
```

```
#<procedure>
```

- Syntax:
  - (All in parentheses)
  - `lambda`
  - Zero or more symbol names in another pair of parenthesis (the function's parameters, if any)
  - One or more expressions (the function's body)
- Value:
  - The value of a lambda expression is a procedure

Language Elements: Abstractions

- Lambda is another special form
- It creates a procedure and returns it as a value
- This value can be used anywhere you would use a procedure:

```
((lambda (x) (* x x)) 5)
```

```
25
```

Application in Action

```
((lambda (x) (* x x)) 5)
```

```
(* 5 5)
```

```
25
```

- We can also give it a name:

```
(define square
  (lambda (x) (* x x)))
```

```
(square 5)
```

 25

“Syntactic Sugar”

Syntactic forms that are more convenient, but don’t make the language any more expressive, are sometimes referred to as “sugar.”

E.g.,

```
(define (square x)
  (* x x))
```

is sugar for:

```
(define square
  (lambda (x) (* x x)))
```

The `if` Special Form

- An `if` expression has three sub-expressions:

```
(if <predicate>
  <consequence>
  <alternative>)
```

- Evaluator first evaluates the predicate expression
- If it evaluates to `#t` (true), then the evaluator evaluates and returns the value of the consequence expression
- Otherwise, it evaluates and returns the value of the alternative expression
- More idiomatic to use `cond`

The `cond` Special Form

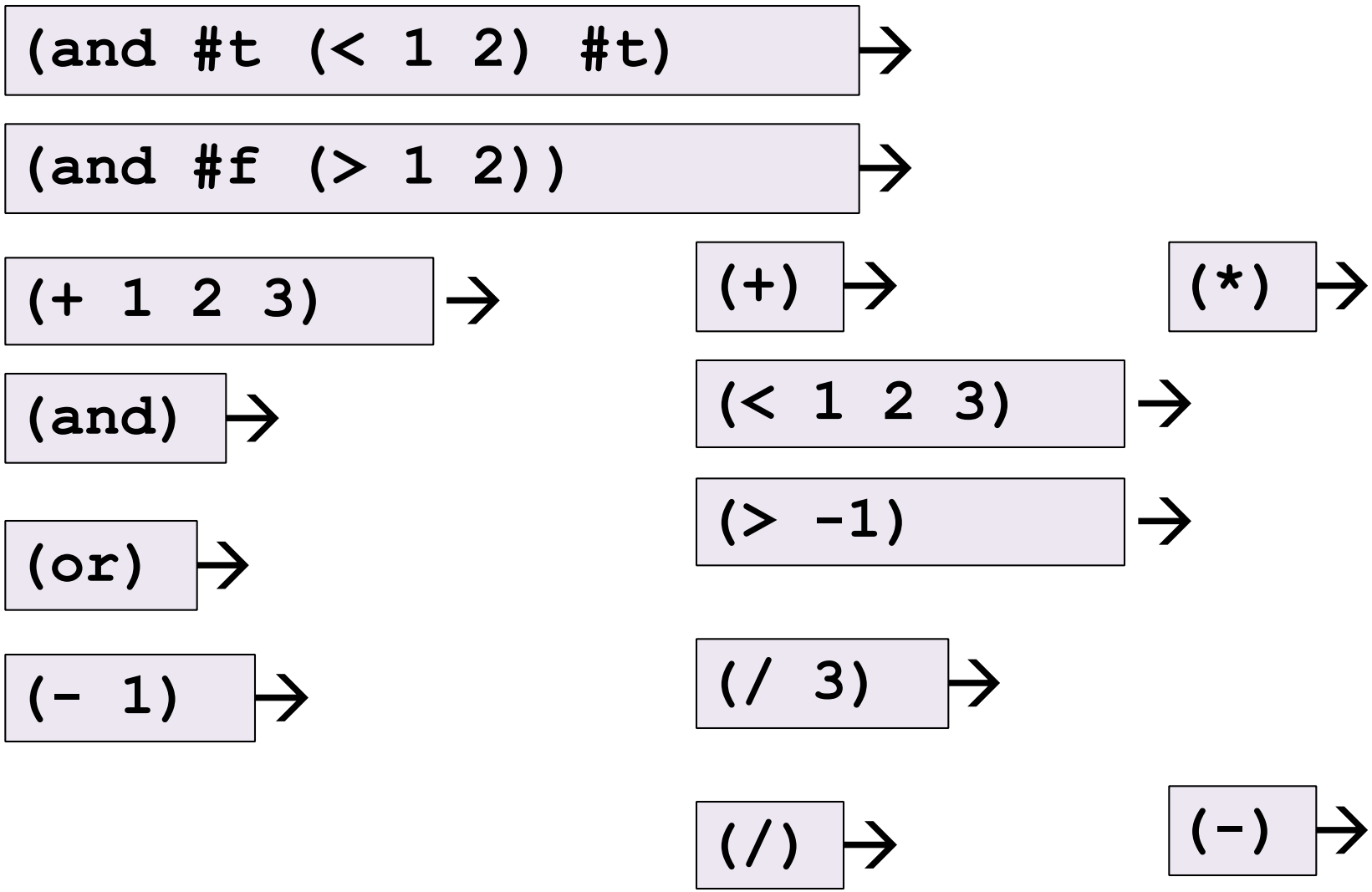
- `if` does not handle *else-if* cases well without lots of nesting. Instead, `cond` can be used:

```
(define (sign-string n)
  (cond
   ((< n 0) "negative")
   ((> n 0) "positive")
   (else "zero")))
```

- If none of the clauses evaluate to true and there is no `else` clause then the result is unspecified

Variable Arity Procedures

Variable-arity procedures can take a different number of arguments. E.g., like `printf` in C.



Other Special Forms

- Both logical and and or ***short circuit:***

(and <e_1> ... <e_n>)

(or <e_1> ... <e_n>)

- Evaluates the expressions one at a time, in left-to-right order.
  - For `and`:
    - if any expr evaluates to false, the value of the `and` is false and the rest of the exprs are not evaluated If true, its value is the value of the last expr
  - For `or`:
    - if any expr evaluates to true, that value is returned and the rest of the exprs are not evaluated
- `not` does not short circuit: it is an ordinary procedure and thus not a special form

The Substitution Model

Rules for expression evaluation in the substitution model:

1. If ***self-evaluating***, (e.g., a number) just return that value.
2. If a ***name***, replace with values associated with that name
3. If expression is a `lambda`, create procedure and return
4. If expression is a special form, (e.g., `if`, `and`) follow specific rules for evaluating sub-expressions
5. If expression is a compound expression, then:
   - Evaluate all of the sub-expressions of combination (in any order)
   - If procedure is primitive, just do it
   - If procedure is compound procedure (created by `lambda`), substitute value of each sub-expression for corresponding procedure parameter in body of procedure, then repeat on body