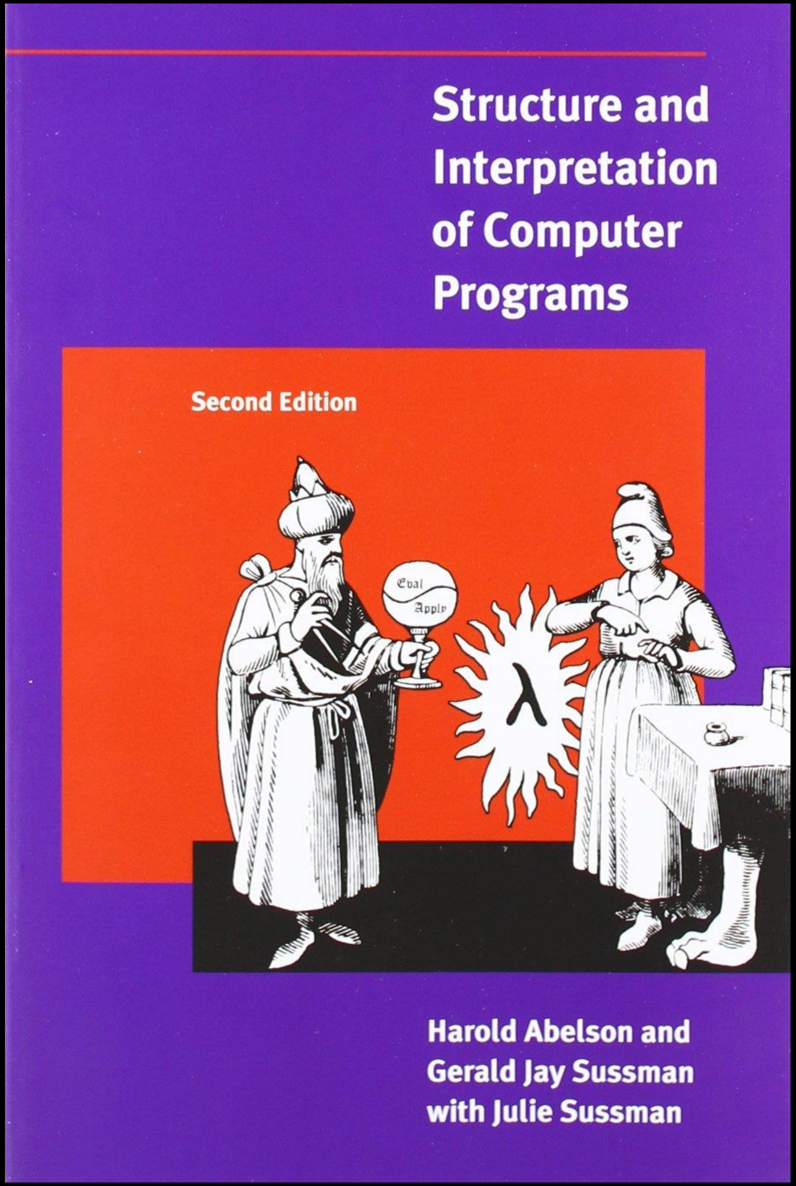
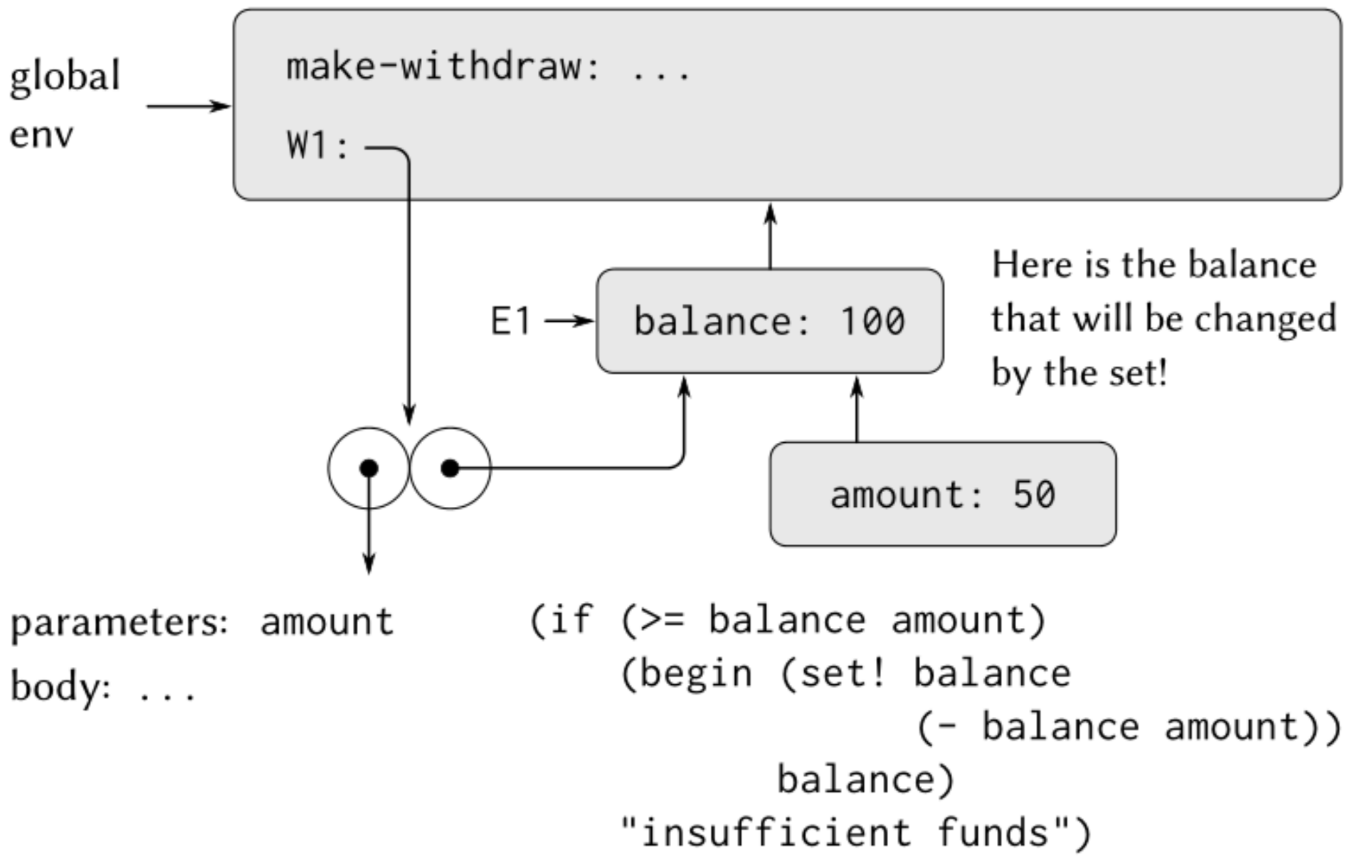
Java

Meetup


Structure and Interpretation of Computer Programs
Second Edition
Eval
Apply
λ
Harold Abelson and Gerald Jay Sussman with Julie Sussman

Structure and Interpretation of Computer Programs

Chaper 3.2 Bonus



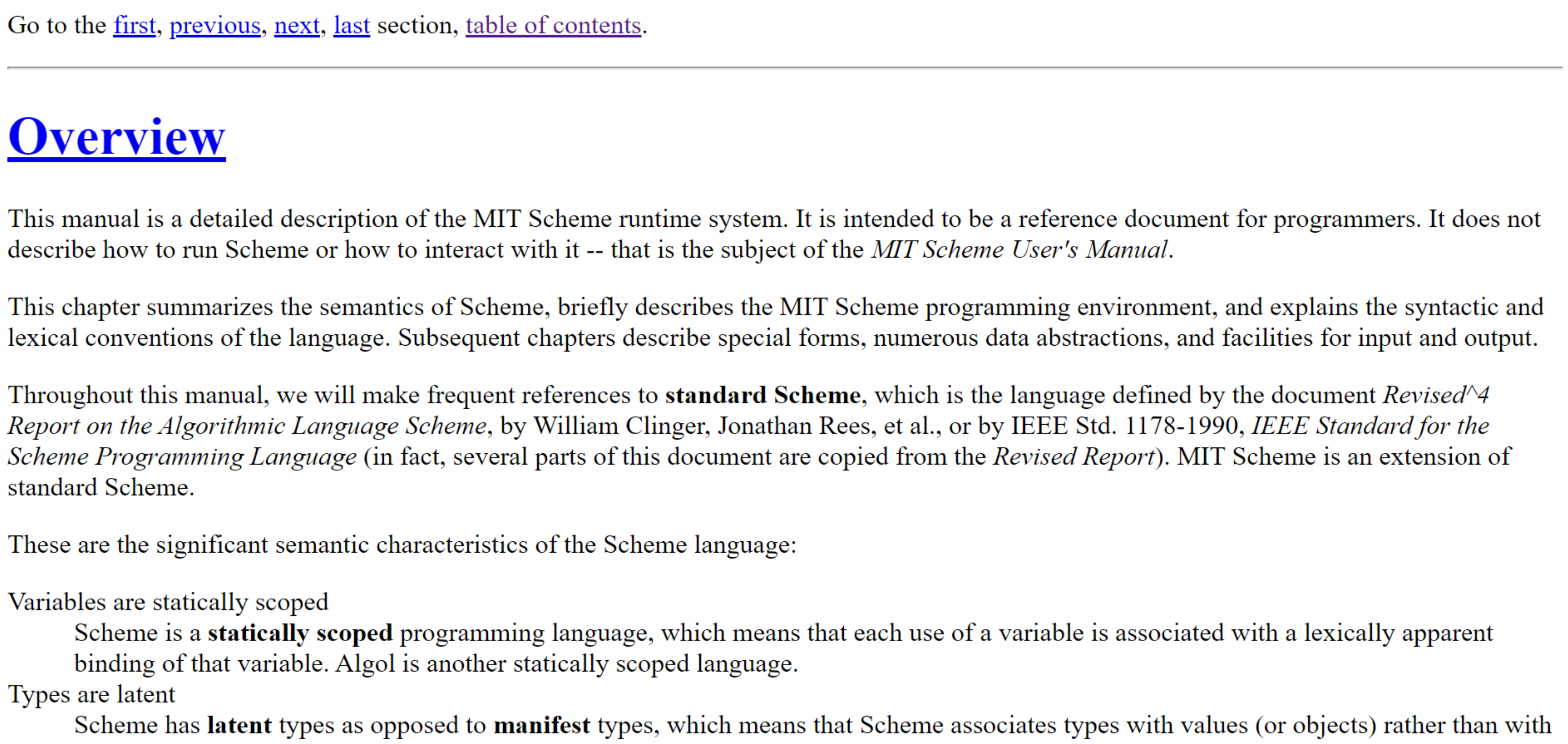
```
(if (>= balance amount)
    (begin (set! balance
                 (- balance amount))
           balance)
    "insufficient funds")
```

Figure 3.8: Environments created by applying the procedure object `W1`.

The environment model of procedure application can be summarized by two rules:

- A procedure object is applied to a set of arguments by constructing a frame, binding the formal parameters of the procedure to the arguments of the call, and then evaluating the body of the procedure in the context of the new environment constructed. The new frame has as its enclosing environment the environment part of the procedure object being applied.
- A procedure is created by evaluating a λ-expression relative to a given environment. The resulting procedure object is a pair consisting of the text of the λ-expression and a pointer to the environment in which the procedure was created.

https://groups.csail.mit.edu/mac/ftpdir/scheme-7.4/doc-html/scheme_2.html



Go to the first, previous, next, last section, table of contents.

Overview

This manual is a detailed description of the MIT Scheme runtime system. It is intended to be a reference document for programmers. It does not describe how to run Scheme or how to interact with it -- that is the subject of the *MIT Scheme User's Manual*.

This chapter summarizes the semantics of Scheme, briefly describes the MIT Scheme programming environment, and explains the syntactic and lexical conventions of the language. Subsequent chapters describe special forms, numerous data abstractions, and facilities for input and output.

Throughout this manual, we will make frequent references to **standard Scheme**, which is the language defined by the document *Revised^4 Report on the Algorithmic Language Scheme*, by William Clinger, Jonathan Rees, et al., or by IEEE Std. 1178-1990, *IEEE Standard for the Scheme Programming Language* (in fact, several parts of this document are copied from the *Revised Report*). MIT Scheme is an extension of standard Scheme.

These are the significant semantic characteristics of the Scheme language:

Variables are statically scoped

    Scheme is a **statically scoped** programming language, which means that each use of a variable is associated with a lexically apparent binding of that variable. Algol is another statically scoped language.

Types are latent

    Scheme has **latent** types as opposed to **manifest** types, which means that Scheme associates types with values (or objects) rather than with

Expressions

A Scheme **expression** is a construct that returns a value. An expression may be a *literal*, a *variable reference*, a *special form*, or a *procedure call*.

Expressions

- A Scheme expression is a construct that returns a value

- There are FOUR types of expressions
  1. Literal (42, “hello world”, etc)
  2. Variable reference (x, name, etc)
  3. Special form (define, lambda, let, cond, if, etc)
  4. Procedure call (o/w known as **combination** or **application**)

Procedure Call Syntax

```
(operator operand ...)
```

A **procedure call** is written by simply enclosing in parentheses expressions for the procedure to be called (the **operator**) and the arguments to be passed to it (the **operands**). The *operator* and *operand* expressions are evaluated and the resulting procedure is passed the resulting arguments. See section Lambda Expressions, for a more complete description of this.

Procedure Call Syntax

```
(operator operand ...)
```

A **procedure call** is written by simply enclosing in parentheses expressions for the procedure to be called (the **operator**) and the arguments to be passed to it (the **operands**). The *operator* and *operand* expressions are evaluated and the resulting procedure is passed the resulting arguments. See section Lambda Expressions, for a more complete description of this.

Another name for the procedure call expression is **combination**. This word is more specific in that it always refers to the expression; "procedure call" sometimes refers to the *process* of calling a procedure.

Unlike some other dialects of Lisp, Scheme always evaluates the operator expression and the operand expressions with the same evaluation rules, and the order of evaluation is unspecified.

```
(+ 3 4)                                 =>  7
((if #f = *) 3 4)                       =>  12
```

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables `+` and `*`. New procedures are created by evaluating `lambda` expressions.

Lambda Expressions

special form: **lambda** *formals expression expression* ...

A `lambda` expression evaluates to a procedure. The

```
;; Procedure syntax
(define (sq x) (* x x)) -> (define sq (λ (x) (* x x)))

;; Let expression syntax
(let ((x y)) ...) -> ((λ (x) (...)) y)
```

1. Invoking a procedure creates a frame ✓
2. Lambda is the value of a procedure ✗
3. You evaluate (lambda-)expressions - not procedures ✓
4. Lambda is an expression ✓
5. Procedure is a value ✓
6. Procedures are applied (A), expressions are evaluated (B) ✓
   ✓