1 Com S 342 so far

So far we have studied:

- Language design goals,
- Basic functional programming,
- Flat recursion over lists,
- Notion of Scheme procedures and closures, and
- Syntax abstraction.

2 Reading

- EOPL 3rd Edition, Section 2.1 - 2.5

3 Specifying Data via Interfaces

The key ideas that we will learn about are data abstraction and abstract data types (ADT). For example: nonnegative integers. Basic idea is to divide program into 2 parts:

1. knows about representation (the ADT implementation)
2. doesn’t know details of representation (the client)

Exercise 3.1 What good is this?

- Allows us to develop data representations gradually.
- Allows us to change the representation by only changing small number of procedures.

3.1 Representation Independence

Definition 3.1 A program exhibits representation independence if it works correctly with any correct implementation of an abstract data type.

Example: most programs are independent of the representation for numbers, lists, we’ll also be independent of the representation of records.

Exercise 3.2 Are C programs independent of the representation of numbers?
For a program to be representation independent, it must rely only on the interface (specification) of the data type. For example, consider the data type of natural numbers. The interface of this data type consists of four procedures defined as follows.

\[
\begin{align*}
\text{(zero)} &= 0 \\
\text{(is-zero? !n!)} &= \text{#t if } n=0 \text{ else } \text{#f} \\
\text{(successor !n!)} &= n+1 \quad (n\geq 0) \\
\text{(predecessor !n+1!)} &= !n! \quad (n>0)
\end{align*}
\]

Given this specification, there could be many implementation that obey it. See the unary, Scheme number and Bignum representations on pages 33-34.

However, if we write client programs that manipulate natural numbers in accordance with these rules, the interface provides a guarantee that they will produce correct answers irrespective of the representation of the natural numbers in use.

**Exercise 3.3** Is there any guarantee if clients manipulate natural numbers in a manner that cannot be expressed in terms of the four procedures above?

### 3.2 Opaque vs. Transparent

**Definition 3.2** A type is **opaque** if there is no way to find out its representation, even by printing. Otherwise it is **transparent**.

**Exercise 3.4** Which kind does Scheme have? C++?

**Exercise 3.5** Which kind of date type does Smalltalk has?

**Exercise 3.6** What’s the advantage of an opaque type? of transparent?

- Opaque enforces use of the defining procedures,
- has better security, and
- transparent is easier to debug and extend (which is also a disadvantage!)

### 4 Representation Strategies for Data Types

So far we looked at different representations of the same data types. Next we will study some of the strategies for representing data types.

Our example will be a data type for **environments**.

**Definition 4.1** An *environment* is a data type that associates a value with each element of a finite set of variables. In other words, it maps variable names to values.
We have mentioned this before, now let us define it concretely!
See Section 2.2.1 for description of the specification/interface of this data type.

An environment’s behavior is defined by the function below:

\[
\begin{align*}
(\text{empty-env}) & = \text{!empty-set!} \\
(\text{apply-env } !f! \text{ var}) & = f(\text{var}) \\
(\text{extend-env } \text{var } v ) !f! & = !g! ,
& \text{ where } g(\text{var1}) = v & \text{ if } \text{var1} = \text{var} \\
& f(\text{var1}) & \text{otherwise}
\end{align*}
\]

**Exercise 4.7** Extend the environment created on page 36 such that the variable `p` is defined as 47.

### 4.1 Data Structure Representation

Note that environment is an expression in the following grammar:

\[
\text{Env} ::= (\text{empty-env}) \\
| (\text{extend-env } \text{Var } \text{SchemeVal } \text{Env})
\]

Review the data-structure based representation in Figure 2.1 on page 38 of the textbook.

```scheme
;;; data definition:
;;; Env ::= (empty-env) | (extend-env Var Schemeval Env)

;;; empty-env : () -> Env
(define empty-env
  (lambda () (list 'empty-env)))

;;; extend-env : Var * Schemeval * Env -> Env
(define extend-env
  (lambda (var val env)
    (list 'extend-env var val env)))

;;; apply-env : Env * Var -> Schemeval
(define apply-env
  (lambda (env search-var)
    (cond
      ;; Case of empty environment
      (eqv? (car env) 'empty-env)
      (report-no-binding-found search-var))

      ;; Case of non-empty environment
      (eqv? (car env) 'extend-env)
      (let ((saved-var (cadr env))
            (saved-val (caddr env))
            (saved-env (cadddr env)))
        (if (eqv? search-var saved-var)
            saved-val
            (apply-env saved-env search-var)))
      (else
        (report-invalid-env env))))
```

### 4.2 Procedural Representation

The key idea here is to use a higher-order procedure to represent the two constructors of the environment: empty-env, extend-env. These procedures store the mapping between variables and values.

```scheme
;;; empty-env : () -> Env
(define empty-env
  (lambda ()
```

3
Review the description of data-structure based representation in Figure 2.1 and on pages 40–41 of the textbook.

4.3 Interfaces for Recursive Data Types

- Why? (Expressions are recursive data types)
- How? (Follow the recipe)
- What? (Constructor, Predicate, and Extractors)
  
  - Include one constructor for each kind of data in the data type.
  - Include one predicate for each kind of data in the data type.
  - Include one extractor for each piece of data passed to a constructor of the data type.

Let us apply this to lambda expressions defined the following grammar.

\[
\text{Lc-exp ::= Identifier "Variable" } \\
| \text{(lambda (Identifier) Lc-exp) "Abstraction" } \\
| \text{(Lc-exp Lc-exp) "Application" }
\]

What are kinds of data for this data type?

- Constructors:
- Predicates:
- Extractors:

Review the general recipe for designing an interface for a recursive data type on page 42–43 of your textbook. Based on this recipe, the interface for the data type lambda expression can be defined as:

\[
\text{;;;Constructors:} \\
\text{;;; var-exp : Var } \to \text{ Lc-exp} \\
(\text{define var-exp} \\
(\text{lambda (var)} \\
('\text{var-exp ,var})))
\]

\[
\text{;;; lambda-exp : Var * Lc-exp } \to \text{ Lc-exp} \\
(\text{define lambda-exp} \\
(\text{lambda (var lc-exp)} \\
('\text{lambda-exp ,var ,lc-exp})))
\]

\[
\text{;;; app-exp : Lc-exp * Lc-exp } \to \text{ Lc-exp} \\
(\text{define app-exp} \\
(\text{lambda (lc-expl lc-exp2)} \\
('\text{app-exp ,lc-expl ,lc-exp2})))
\]
4.4 A Tool for Defining Data Types

Why do we need such a tool?

- For complicated data types, manual definition tedious
- For example, if a data type has n variants and each variant has m fields, the number of procedures that we have to define are _______?
- Avoid errors in data type definitions

How do we come up with such a tool?

- Defining data type interfaces is fairly systematic
- In fact, we looked at an algorithm for it
  - Defines Constructors, Predicates, Observers
- So why not use syntax abstraction and abstract away from this abstraction
  - Is it applicable? Does syntax abstraction precisely serves this purpose?

Let us revisit the data type for lambda expressions as defined previously.

;;; The form is "define-datatype"
;;; followed by "type name"
;;; followed by "predicate name" for the type.
(define-datatype lc-exp lc-exp
  ;;; Predicates:
  ;;; var-exp? : Lc-exp -> Bool
  (define var-exp?
    (lambda (x)
      (and (pair? x) (eq? (car x) 'var-exp))))
  ;;; lambda-exp? : Lc-exp -> Bool
  (define lambda-exp?
    (lambda (x)
      (and (pair? x) (eq? (car x) 'lambda-exp))))
  ;;; app-exp? : Lc-exp -> Bool
  (define app-exp?
    (lambda (x)
      (and (pair? x) (eq? (car x) 'app-exp))))
  ;;; Extractors:
  ;;; var-exp->var : Lc-exp -> Var
  (define var-exp->var
    (lambda (x) (cadr x)))
  ;;; lambda-exp->bound-var : Lc-exp -> Var
  (define lambda-exp->bound-var
    (lambda (x) (cadr x)))
  ;;; lambda-exp->body : Lc-exp -> Lc-exp
  (define lambda-exp->body
    (lambda (x) (caddr x)))
  ;;; app-exp->rator : Lc-exp -> Lc-exp
  (define app-exp->rator
    (lambda (x) (cadr x)))
  ;;; app-exp->rand : Lc-exp -> Lc-exp
  (define app-exp->rand
    (lambda (x) (caddr x)))

Note the three type of procedures: constructors, predicates and extractors.
A case for each variant/kind of data type. [At least one kind is required]

Each case has the form "variant name" followed by a "record type".

A record type contains zero or more field declarations
A field declaration has the form (<field-name> <predicate>)

Here is the case for variable
(var-exp (var identifier?) )

Here is the case for abstraction
(lambda-exp (bound-var identifier?) (body lc-exp?) )

Here is the case for application
(app-exp (rator lc-exp?) (rand lc-exp?) )

Following are the effect of this define-datetype definition.

- Creates a date-type with specified variants
- Each variant has a variant-name and 0 or more fields
- Each field has its own name and associated predicate
- Creates a new constructor for each variant.
- Bounds the type predicate name to a predicate, which determines if its argument is a value of the type.
- Constraints:
  - No two types may have the same name.
  - No two variants have the same name.
  - Type names may not used as variant names.
  - Each field predicate must be a Scheme predicate.

As a result of the date type definition above, we now have access to constructors, predicates and extractors. So now we can write the occurs-free function of Chapter 1 in terms of these procedures as shown below. See your textbook for more description on this.

Objective is to find if search-var is a free variable in exp.
(define occurs-free? (lambda (search-var exp)
    (cases lc-exp exp
        (var-exp (var) (eqv? var search-var))
        (lambda-exp (bound-var body)
            (and
                (not (eqv? search-var bound-var))
                (occurs-free? search-var body)))
    ))
;;; Read it as: "If exp was of kind app-exp then it would have two fields and ..."
(app-exp (rator rand)
or
  (occurs-free? search-var rator)
  (occurs-free? search-var rand))))))

;;; Additional facts: can have a default case, positional binding.

Some more usage can be seen in Section 2.5 in your textbook. Note that entire chapter 2 is on your required reading list for this course.