Complex Numbers

- We will develop a system that performs complex number operations.
- We will use two representations: Rectangular (real + imaginary part), and polar (magnitude and angle).
- Complex numbers are pairs, just like the rational number example we saw earlier.
- A complex number $z = x + iy$ where $i = \sqrt{-1}$ can be thought of as a point in an $x, y$ plane.
- The polar form is $z = re^{iA}$ where $r$ is the magnitude and $A$ is the angle with the $x$ axis.
Converting Between Representations

\[ z = x + iy = re^{iA} \]

Diagram:
- Real axis labeled \( x \)
- Imaginary axis labeled \( y \)
- Point \( z \) is represented by \( r \) and \( \theta \) in polar coordinates.

Nurit Haspel  
CS450 - Structure of Higher Level Languages
Given $x, y, r, A$

- $x = r \cos A$
- $y = r \sin A$
- $r = \sqrt{x^2 + y^2}$
- $A = \text{atan}(y, r)$

By convention, $A$ is the angle with the $x$ axis, so $A = 0^\circ$ when aligned with the $x$ axis.
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z)
  (sqrt (+ (square (real-part z)) (square (imag-part z)))))
(define (angle z)
  (atan (imag-part z) (real-part z)))
(define (make-from-real-imag x y) (cons x y))
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))
(define (real-part z)  
  (* (magnitude z) (cos (angle z))))
(define (imag-part z)  
  (* (magnitude z) (sin (angle z))))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-real-imag x y)  
  (cons (sqrt (+ (square x) (square y)))  
        (atan y x)))
(define (make-from-mag-ang r a) (cons r a))
Operations on Complex Numbers

- **Addition:**
  - $\text{Real}(z_1 + z_2) = \text{Real}(z_1) + \text{Real}(z_2)$
  - $\text{Imaginary}(z_1 + z_2) = \text{Imaginary}(z_1) + \text{Imaginary}(z_2)$

- **Multiplication (more convenient to use polar):**
  - $\text{Magnitude}(z_1 \cdot z_2) = \text{Magnitude}(z_1) \cdot \text{Magnitude}(z_2)$
  - $\text{Angle}(z_1 \cdot z_2) = \text{Angle}(z_1) + \text{Angle}(z_2)$
We want both representation to be available to us.
We want all operations to be available regardless of which representation we are using.
Assume we have four selectors: real-part, imag-part, magnitude and angle
Assume we have two constructors: make-from-real-imag and make-from-mag-angle.
Given a complex number \( z \), both constructors should return complex numbers that are equal to \( z \).
See sec2.4.1.scm.pdf for operations on complex numbers.
Data abstraction allows us to use either of the two representations above.

As a matter of fact, we can even use both!

It is great, since rectangular representation goes more naturally with some operations, and polar goes more naturally with others.

However, we need to distinguish the data in polar form from the data in rectangular form.

Otherwise, given two numbers, we wouldn’t know if they are the real and imaginary or the magnitude and angle.

To accomplish that, we can attach a type tag to our data.
Assume that we have procedures type-tag and contents that extract from a data object the tag and the actual contents (the polar or rectangular coordinates, in the case of a complex number).

Additionally, a procedure attach-tag takes a tag and contents and produces a tagged data object.
(define (attach-tag type-tag contents)
  (cons type-tag contents))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum -- TYPE-TAG" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum -- CONTENTS" datum)))
Now we can use both representations in the same package.

We define predicates `rectangular?` and `polar?` to identify which one we’re using.

```
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
(define (polar? z)
  (eq? (type-tag z) 'polar))
```
When building new procedures, we should remember to name our functions uniquely and to attach tags.

```
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
          (square (imag-part-rectangular z)))))
(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
               (cons (* r (cos a)) (* r (sin a))))))
```
When building new procedures, we should remember to name our functions uniquely and to attach tags.

(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z))))
(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z))))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
    (cons (sqrt (+ (square x) (square y)))
    (atan y x))))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))
Check the tag to know which implementation to use.

(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
       (else (error "Unknown type -- REAL-PART" z))))

(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
       (else (error "Unknown type -- IMAG-PART" z)))))
Usage Example

Check the tag to know which implementation to use.

```
(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))
        (else (error "Unknown type -- MAGNITUDE" z))))
```

```
(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))
        (else (error "Unknown type -- ANGLE" z))))
```
For arithmetic operations, use the same procedures as before. The reason is that the selectors are generic, and they decide which representation they work with. There are several layers of abstraction here. The tags are needed for the higher level procedures, to recognize what representation they are using.

Programs that use complex numbers

add-complex sub-complex mul-complex div-complex

Complex arithmetic package

real-part imag-part magnitude angle

Rectangular Representation Polar Representation

List structure and primitive machine arithmetics
The system above has some weaknesses.

For one, every representation needs to know about the others.

Imagine we add a third representation...

Also, we have to make sure no two procedures have the same name.

This method is not *additive*: The person implementing the generic selector procedures must modify those procedures each time a new representation is installed.

The people interfacing the individual representations must modify their code to avoid name conflicts.
Notice that whenever we deal with a set of generic operations that are common to a set of different types we are dealing with a two-dimensional table that contains the possible operations on one axis and the possible types on the other axis.

The entries in the table are the procedures that implement each operation for each type of argument presented.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Polar</th>
<th>Rectangular</th>
</tr>
</thead>
<tbody>
<tr>
<td>real-part</td>
<td>real-part-polar</td>
<td>real-part-rectangular</td>
</tr>
<tr>
<td>imag-part</td>
<td>imag-part-polar</td>
<td>imag-part-rectangular</td>
</tr>
<tr>
<td>magnitude</td>
<td>magnitude-polar</td>
<td>magnitude-rectangular</td>
</tr>
<tr>
<td>angle</td>
<td>angle-polar</td>
<td>angle-rectangular</td>
</tr>
</tbody>
</table>
Data-directed programming works with such a table directly.

Here we will implement the interface as a single procedure that looks up the combination of the operation name and argument type in the table, and then applies it to the contents of the argument.

This way, adding a new representation package to the system only requires adding new entries to the table.

(put op type item): install an item in the table, indexed by op, type

(get op type): Retrieve an item from the table, indexed by op, type.

For now assume these functions exist in our language.
How Does it Work?

- We develop our code as usual.
- Define a collection of procedures, or a \textit{package} and interfaces to the rest of the system by adding entries to the table:

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z) (sqrt (+ (square (real-part z)) (square (imag-part z)))))
  (define (angle z) (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a) (cons (* r (cos a)) (* r (sin a))))
```
interface to the rest of the system

(define (tag x) (attach-tag 'rectangular x))

(put 'real-part '(rectangular) real-part)
(put 'imag-part '(rectangular) imag-part)
(put 'magnitude '(rectangular) magnitude)
(put 'angle '(rectangular) angle)
(put 'make-from-real-imag 'rectangular
 (lambda (x y) (tag (make-from-real-imag x y))))
(put 'make-from-mag-ang 'rectangular
 (lambda (r a) (tag (make-from-mag-ang r a))))
'done)
The internal procedures here are the same as before.

No changes are necessary in order to interface them to the rest of the system.

Moreover, since these procedure definitions are internal to the installation procedure, there is no need to worry about name conflicts.

For polar package see the text. It’s very similar.
The complex-arithmetic selectors access the table by means of a general operation” procedure called apply-generic, which applies a generic operation to some arguments.

It searches the table under the name of the operation and the types of the arguments and applies the resulting procedure if one is present:

```scheme
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error "No method for these types -- APPLY GENERIC" (list op type-tags)))))))
```
We can define our generic selectors as follows:

\[
\begin{align*}
&\text{(define (real-part z) (apply-generic 'real-part z))} \\
&\text{(define (imag-part z) (apply-generic 'imag-part z))} \\
&\text{(define (magnitude z) (apply-generic 'magnitude z))} \\
&\text{(define (angle z) (apply-generic 'angle z))}
\end{align*}
\]

This way we can add new definitions without changing the old ones.
We can also extract from the table the constructors to be used by the programs in making complex numbers from real and imaginary parts and from magnitudes and angles.

We construct rectangular numbers whenever we have real and imaginary parts, and polar numbers whenever we have magnitudes and angles:

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))
(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```
This style of programming organizes the required dispatching on type by having each operation take care of its own dispatching.

This decomposes the operation-and-type table into rows, with each generic operation procedure representing a row of the table.

An alternative strategy is to decompose the table into *columns* and, instead of using “intelligent operations” that dispatch on data types, to work with “intelligent data objects” that dispatch on operation names.

We can do this by arranging things so that a data object, such as a rectangular number, is represented as a procedure that takes as input the required operation name and performs the operation indicated.
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude)
            (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else
           (error "Unknown op -- MAKE-FROM-REAL-IMAG" op)))
    dispatch)

The corresponding apply-generic procedure is now as follows:

(define (apply-generic op arg) (arg op))
This style of programming is called message passing.
The name comes from the image that a data object is an entity that receives the requested operation name as a “message”.
We have seen it before with our possible implementation of cons.
We will get back to this idea later on in the course.