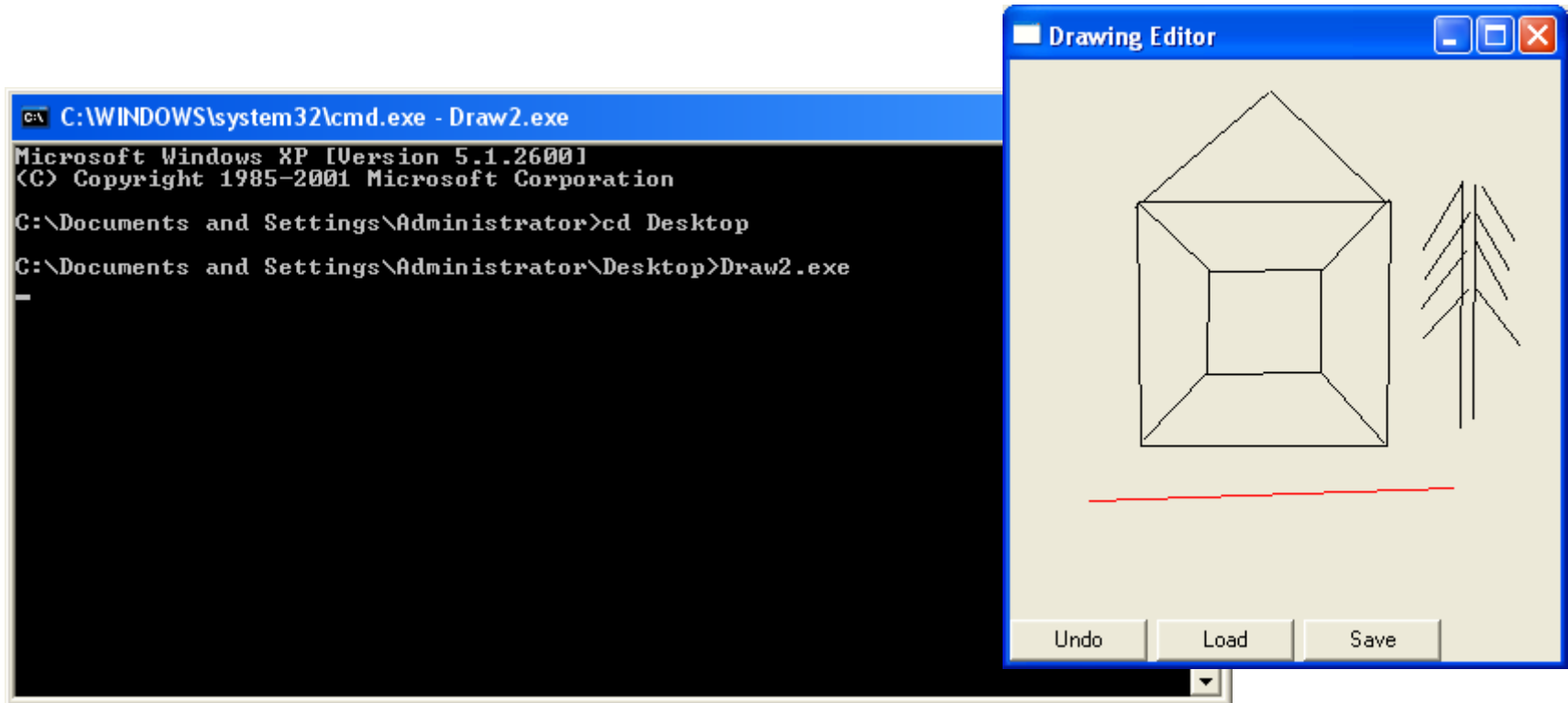




Programming IO

Let's run a Haskell program...



- What's the type of *that* result???

PATIENCE! We'll program GUIs in a few weeks.

A Much Simpler Example

```
Prelude> writeFile "foo" "baz"
```

```
Prelude>
```

- Writes baz to the file called foo.
- No result displayed—wonder why not?

What's "foo"?

```
Prelude> :t "foo"  
"foo" :: [Char]  
Prelude> :i String  
-- type constructor  
type String = [Char]
```

Huh? I
thought it
was a String

A type synonym

- A String is a *list of characters*
- A character (Char) corresponds more-or-less to a key on the keyboard.
- Examples: 'a', '1', ''

What's writeFile?

```
Prelude> :i writeFile  
writeFile :: FilePath -> String -> IO ()
```

Just a
String

INSTRUCTIONS to
the operating system
to write the file

- When GHCi finds an expression of IO type, it *obeys the instructions* instead of printing them.

An Analogy

- Instructions:

Take this card, go to a Bankomat.
Put in the card.
Enter this code, select 500kr.
Take the money and the card.

- Value:



Which would
you rather
have?

Instructions with Results

- Instructions can have results:

```
Prelude> :i readFile  
readFile :: FilePath -> IO String
```

Instructions for
computing a String

- `readFile "foo"` *is not a String*, and no String can be extracted from it
- But it can be used as *part* of more complex instructions, to compute a String

Just as no 500:- can be
extracted from a bank card

Combining Instructions

- We combine instructions using **do**:

```
copyFile fromA toB =  
  do contents <- readFile fromA  
      writeFile toB contents
```

”First follow
readFile
instructions,
call the result
contents,
then follow
writeFile
instructions”

- `readFile fromA` is an **IO String**
- But `contents` is just a **String**
- ~~`writeFile toB (readFile fromA)`~~

Example: Displaying Instruction Results

```
display io =  
  do result <- io  
    print result
```

Follow the instructions to get a value, then print it

```
Main> display (readFile "foo")
```

```
"baz"
```

```
Main> display (writeFile "foo" "bar")
```

```
()
```

Repeating Instructions

```
doTwice io =  
  do a <- io  
     b <- io  
     return (a,b)  
dont io =  
  return ()
```

An instruction to
compute the given
result

```
Main> display (doTwice (print "hello"))
```

```
"hello"
```

```
"hello"
```

```
((),())
```

```
Main> display (dont (print "hello"))
```

```
()
```

Writing instructions and *obeying*
them are two different things!

Why Distinguish Instructions?

- *Functions* always give the same result for the same arguments
- *Instructions* can behave differently on different occasions
- Confusing them (as in most programming languages) is a major source of bugs
 - This concept a major breakthrough in programming languages in the 1990s
 - How would you write doTwice in C?

Monads = Instructions

- What is the type of doTwice?

```
Main> :i doTwice  
doTwice :: Monad a => a b -> a (b,b)
```

Even the *kind of instructions* can vary!
Different kinds of instructions, depending on who obeys them.

Whatever kind of result argument produces, we get a pair of them

IO means operating system.

Monads = Instructions

- A new built-in type

Instructions to the
Operating System

- IO a

- Standard functions:

- putStr :: String -> IO ()

() is the "empty tuple"
– no interesting
contents

- readFile :: FilePath -> IO String

- writeFile :: FilePath -> String -> IO ()

- ...

Combining Instructions

- Create larger instructions from smaller ones
- Only instructions on top-level are executed!

```
revAndSort :: FilePath -> IO Int
revAndSort file =
  do s <- readFile file
     writeFile ("reversed-" ++ file) (reverse s)
     writeFile ("sorted-" ++ file) (sort s)
     return (length s)
```

An example

- Suppose:
 - lastCommand :: [IO a] -> IO a
 - lastCommand ios = head (reverse ios)
- What happens:
 - lastCommand [print "apa", print "bepa", print "cepa"]

Sequence

- Useful functions:

`sequence :: [IO a] -> IO [a]`

`sequence_ :: [IO a] -> IO ()`

- Example:

`printTable :: [String] -> IO ()`

`printTable xs = ?`

`ghci> printTable ["apa", "bepa", "cepa"]`

1: apa

2: bepa

3: cepa

printTable

```
printTable :: [String] -> IO ()
printTable xs = sequence_
  [putStrLn (show i ++ ":" ++ x)
  |(x,i) <- zip xs [1..]
  ]
```

printTable

Or equivalently:

```
printTable :: [String] -> IO ()
printTable xs =
    sequence_ (map putStrLn table)
  where table = [(show i ++ ":" ++ x)
                | (x,i) <- zip xs [1..] ]
```

Reading

- About I/O: Chapter 18 of the text book
- About QuickCheck: read the *manual* linked from the course web page.
 - There are also several research papers about QuickCheck, and advanced tutorial articles.