University of Auckland

415.360

Functional Programming

Christian Collberg
March 1, 1996

Introducing Gofer

What is Gofer? I

- G. is a functional programming language. It is similar to (and an extended subset of) another FP language, Haskell.
- ∃ free implementations for Unix, PCs, and Macs. They are very similar, and you may (probably) use whichever one you like.
- The G. implementations are **Interpreted** which means that the G. code you write is translated into a high-level intermediate code rather than into machine instructions (the way most C-compilers do). The intermediate code is then executed by a program called the **interpreter**. This is slow.

Slide 2–1

What is Gofer? II

- G. implementations are also **Interactive** which means that the user interface is like a **calculator**; you enter expressions, the Gofer interpreter checks them, evaluates them, and prints the result. This is called the "read-eval-print" loop:

Read → Eval → Print

Example:

```
> gofer
Gofer Version 2.30 ...
Reading script file "standard.prelude":
Gofer session for:
/usr/local/gofer/lib/standard.prelude
?  2*(5+3)
16
(5 reductions, 10 cells)
?
```

Slide 2–2

What is Gofer? III

- You can also write more complicated programs and load them into Gofer. Programs are stored in normal text-files, and consist (mostly) of function definitions.

Example: Eliza

```
?  :load /usr/local/gofer/demos/Eliza/eliza.gs
?  main
Hi!  I'm Eliza, your personal therapy computer.
Please tell me your problem.
I hate you
We were discussing you --not me.
I love my mother.
I see.
I love my mother.
Why did you repeat yourself?
i want you
Oh, you want me?
yes
You seem quite positive.
no
Are you saying no just to be negative?
```

Slide 2–3

Slide 2–4

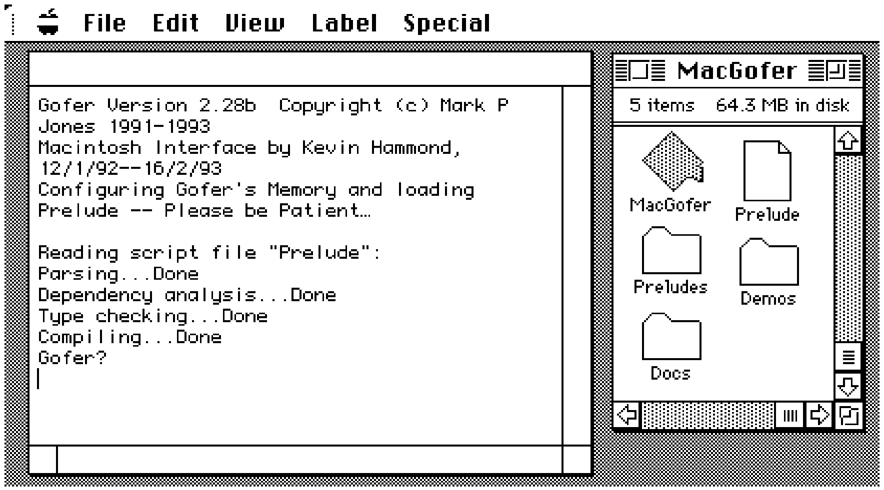
What is Gofer? III

```
eliza :: Dialogue
eliza = interact (("\n\
                  \Hi! I'm Eliza. I am your personal therapy computer.\n\
                  \Please tell me your problem.\n\n" ++)
                 . session initial []
                 . filter (not.null)
                 . map (words . trim)
                 . lines)

session                 :: State -> Words -> [Words] -> String
session rs prev []      = []
session rs prev (l:ls) = response ++ "\n\n" ++ session rs' l ls
                         where (response, rs') | prev == l = repeated rs
                                               | otherwise = answer rs l
```

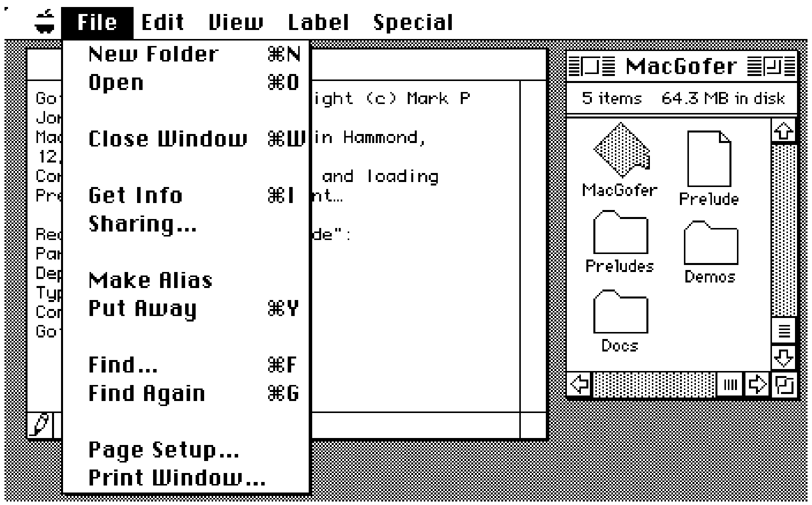
Slide 2–5

Using MacGofer I



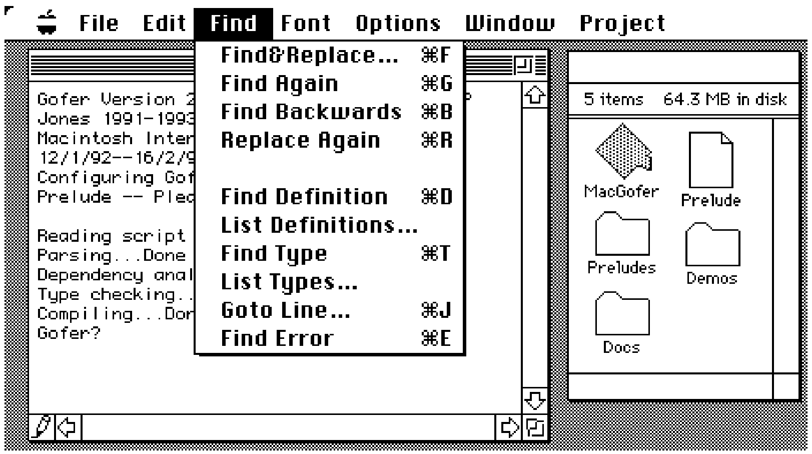
Slide 2–6

Using MacGofer II – File



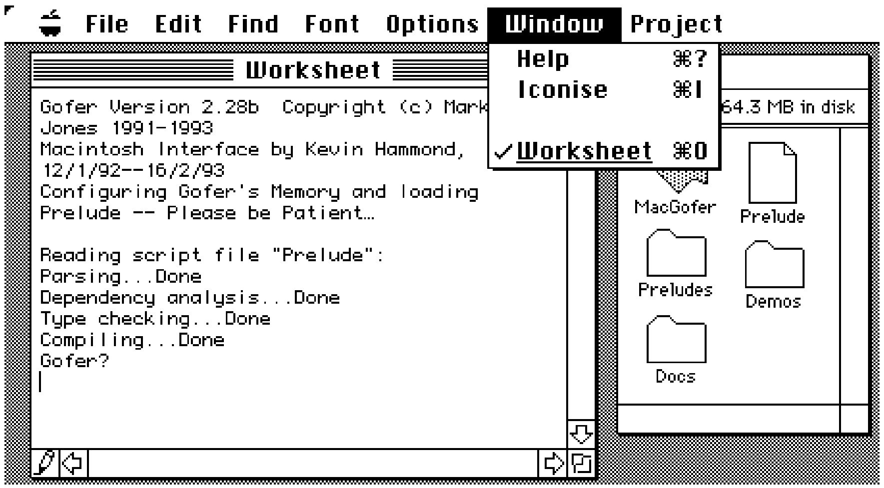
Slide 2–7

Using MacGofer III – Find



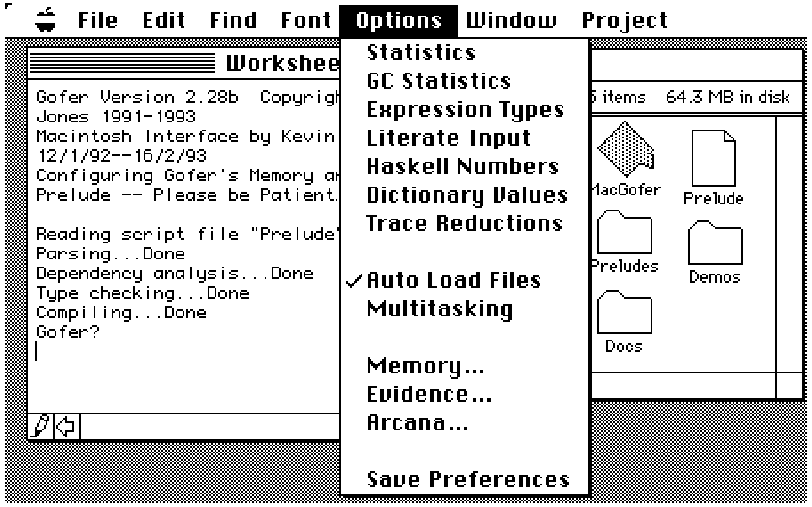
Slide 2–10

Using MacGofer VI – Window



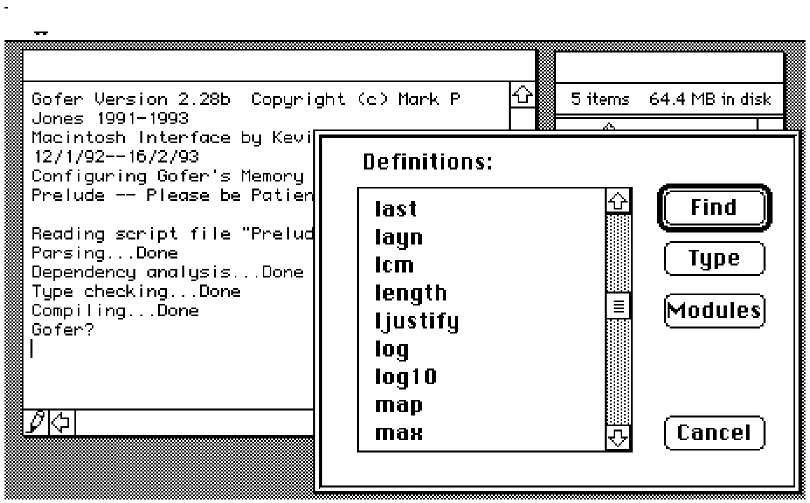
Slide 2–8

Using MacGofer IV – Options



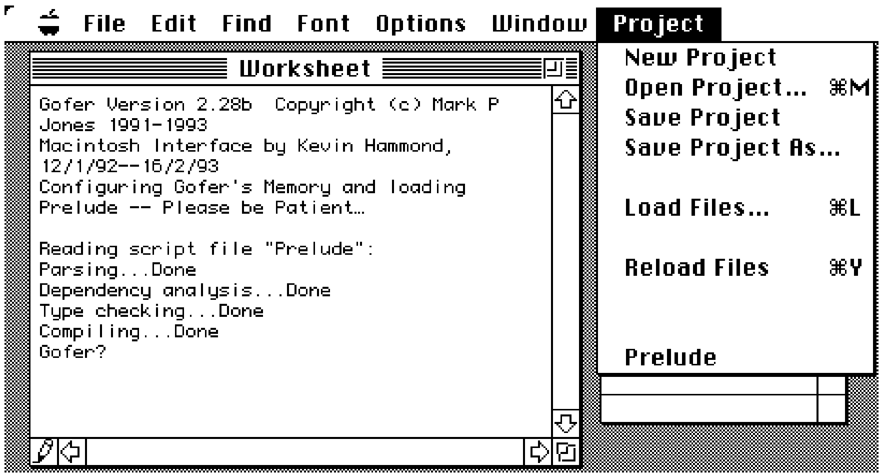
Slide 2–11

Using MacGofer VII – Find Definitions

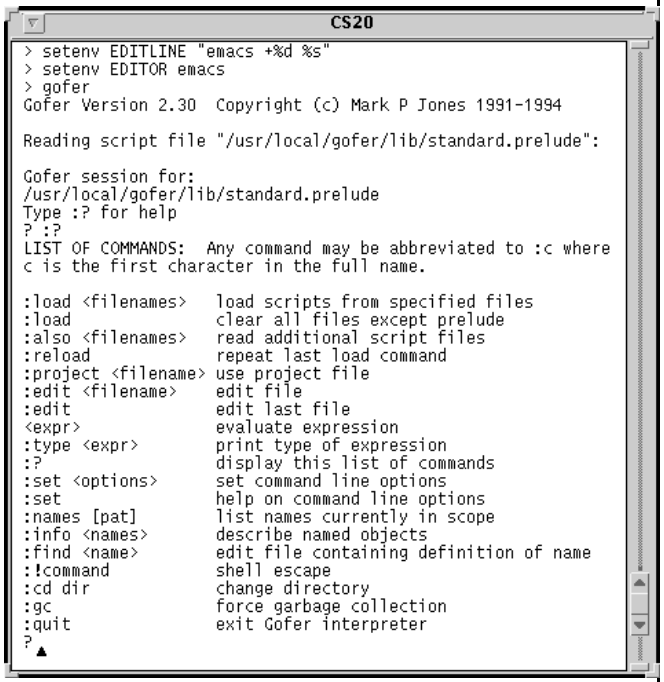


Slide 2–9

Using MacGofer V – Project

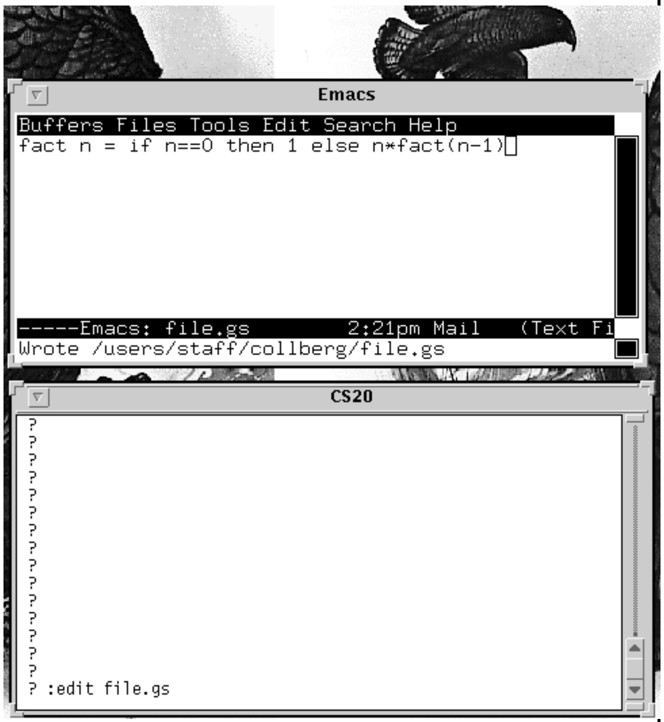


Using Unix Gofer I – Starting Up



Slide 2–12

Using Unix Gofer II – Editing

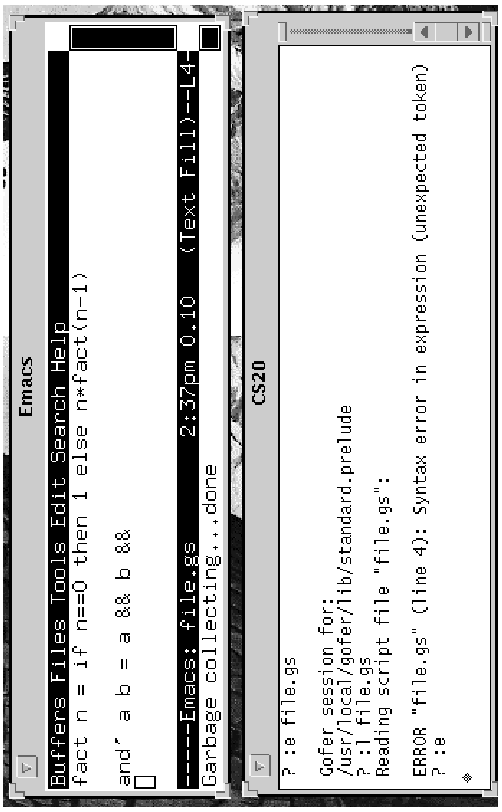


Slide 2–13

Using Unix Gofer III – Loading

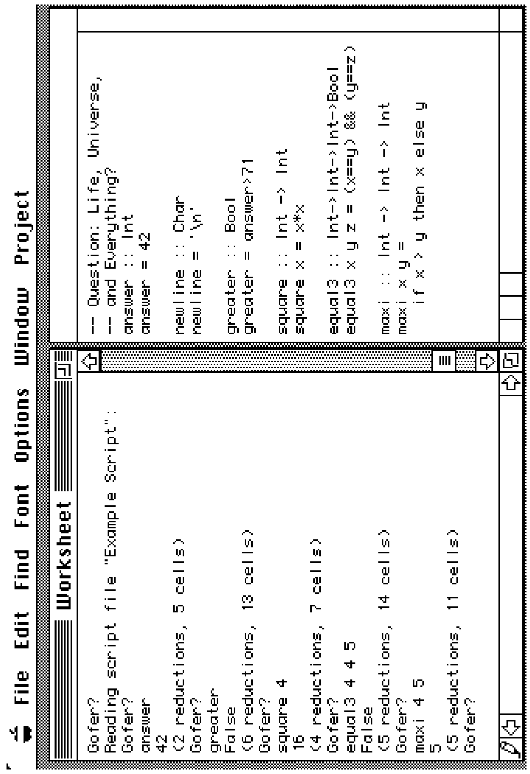
Slide 2–14

Using Unix Gofer IV – Errors



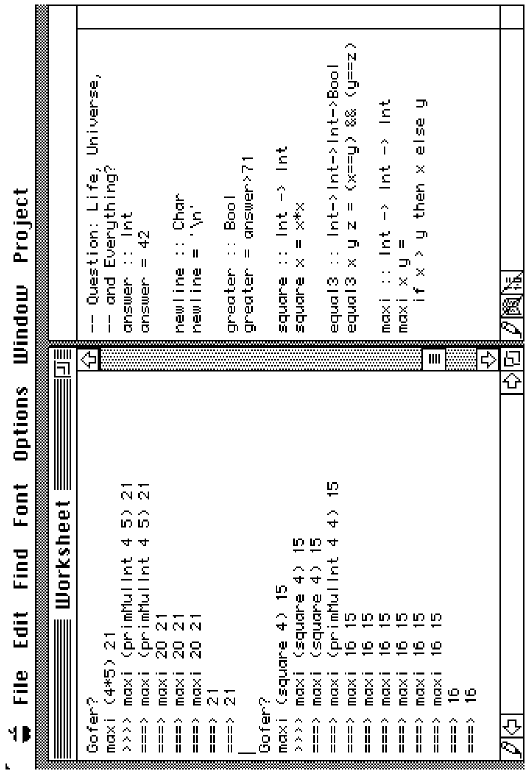
Slide 2–15

A Simple Gofer Script I



Slide 2–16

A Simple Gofer Script II – Tracing



Slide 2–17

`commaint` – A Gofer Program I

- So what does a "real" functional Gofer program look like? Let's have a quick look at one simple (?) function, `commaint`.
- `commaint` works on strings, which are simply lists of characters.
- You are not supposed to understand this! Yet...

From the `commaint` documentation:

> [`commaint`] takes a single string argument containing a sequence of digits, and outputs the same sequence with commas inserted after every group of three digits, . . .

Slide 2–18

`commaint` – A Gofer Program II

Sample interaction:

```
?  commaint "1234567"
   1,234,567
```

`commaint` simulation:

```
"1234567"
    |      reverse
    v
"7654321"
    |       iterate (drop 3)                        g
    v                                               r
["7654321","4321","1","","", ...]                   o
    |       map (take 3)                            u
    v                                               p
["765","432","1","","",...]
    |       takeWhile (not.null)                    3
    v
["765", "432", "1"]
    |       foldr1 (\x y->x++","++y)
    v
"765,432,1"
    |       reverse
    v
"1,234,567"
```

Slide 2–19

Slide 2–20

`commaint` – A Gofer Program III

`commaint` in Gofer:

```
commaint = reverse .  foldr1 (\x y->x++","++y) .
           group 3 .  reverse
           where group n =takeWhile (not.null) .
                              map (take n) .  iterate (drop n)
```

`commaint` in English:

"First reverse the input string. Take the resulting string and separate into chunks of length 3. Then append the chunks together, inserting a comma between chunks. Reverse the resulting string."

Slide 2–21

`commaint` – A Gofer Program IV

```
commaint = reverse.foldr1(\x y->x++","++y).group 3.reverse
   where group n = takeWhile(not.null).map(take n).iterate(drop n)
```

- `group n` is a "local function." It takes a string and an integer as arguments. It divides the string up in chunks of length `n`.
- `reverse` reverses the order of the characters in a string.
- `drop n xs` returns the string that remains when the first `n` characters of `xs` are removed.

Slide 2–22

`commaint` – A Gofer Program V

```
commaint = reverse.foldr1(\x y->x++","++y).group 3.reverse
   where group n = takeWhile(not.null).map(take n).iterate(drop n)
```

- `iterate (drop 3) s` returns the infinite (!) list of strings

  `[s, drop 3 s, drop 3 (drop 3 s),`
  `drop 3 (drop 3 (drop 3 s)), ···]`
- `take n s` returns the first `n` characters of `s`.

Slide 2–23

`commaint` – A Gofer Program VI

```
commaint = reverse.foldr1(\x y->x++","++y).group 3.reverse
   where group n = takeWhile(not.null).map(take n).iterate(drop n)
```

- `map (take n) s` takes a list of strings as input. It returns another list of strings, where each string has been shortened to `n` characters. `(take n)` is a function argument to `map`.
- `takeWhile (not.null)` removes all empty strings from a list of strings.
- `foldr1 (\x y->x++","++y) s` takes a list of strings `s` as input. It appends the strings together, inserting a comma inbetween each pair of strings.

Slide 2–24

`commaint` – A Gofer Program VII

- Since Gofer is an interactive language, we can always try out (parts of) functions that we don't understand.

```
?  reverse "1234567"
   7654321
?  take 3 "dasdasdasd"
   das
?  map (take 3) ["1234","23423","45324",""]
   ["123", "234", "453", []]
?  iterate (drop 3) "7654321"
   ["7654321", "4321", "1", [], [], ··· {interrupt!}
```

Slide 2–25

Basic Types I (a) – Int

- Like we've seen, Gofer supports the integer (`Int`) type.
- Note that in B&W `Int` is `int`.

Integer Operators:

Op	Precedence	Associativity	Description
`^`	8	right	Exponentiation
`*, /`	7	left	Mul, Div
`div`	7	free	Division
`rem`	7	free	Remainder

Slide 2–26

Basic Types I (b) – Int

Op	Precedence	Associativity	Description
`mod`	7	free	Modulus
`+, -`	6	left	Add, Subtract
`==,/=`	4	free	(In-) Equality
`<,<=,>,>=`	4	free	Relational Comparison

`1+2-3`	⇒ `(1+2)-3`	`4==5==6`	⇒ $\boxed{\texttt{ERROR}}$
`1+2*3`	⇒ `1+(2*3)`	`12/6/3`	⇒ $\boxed{\texttt{ERROR}}$
`2^3^4`	⇒ `2^(3^4)`	`12/(6/3)`	⇒ `6`

Slide 2–27

Basic Types II – Bool

- There are two boolean literals, `True` and `False`

Op	Precedence	Associativity	Description
`&&`	3	right	logical and
`||`	2	right	logical or
`not`	9	–	logical not

`3 < 5 && 4 > 2` ⇔ `(3 < 5) && (4 > 2)`

`True || False && True` ⇔ `True || (False && True)`

Gofer Functions I

- Here's the ubiquitous factorial function:

```
fact ::  Int -> Int
fact n =    if n == 0 then
                1
            else
                n * fact (n-1)
```

- The first part of a function definition is the **type signature**, which gives the **domain** and **range** of the function:

```
fact ::  Int -> Int
```

- The second part of the definition is the **function declaration**, the implementation of the function:

```
fact n = if n == 0 then ···
```

Slide 2–28

Gofer Functions II

- The syntax of a type signature is

```
fun_name ::  arg_types
```

`fact` takes one integer input argument and returns one integer result.

- The syntax of function declarations:

```
fun_name param_names = fun_body
```

- `fact` is defined recursively, i.e. the function body contains an application of the function itself.
- Function application examples:

```
fact 1      ⇒ 1
fact 5      ⇒ 120
fact (3+2)  ⇒ 120
```

Slide 2–29

Basic Types III - Char

- Literals: `'a'`, `'b'`. Special characters: `'\n'` (newline), `'\t'` (tab).
- ASCII: `'\65'` (decimal), `'\0101'` (octal), `'\x41'` (hex).
- `toUpper`, `isAlpha`, etc, are defined in the **standard prelude**.

Built-in Functions:

```
ord ::  Char -> Int
char ::  Int -> Char
toUpper, toLower ::  Char -> Char
isAscii,isDigit,··· ::  Char -> Bool
isUpper,isLower,··· ::  Char -> Bool
```

Examples:

```
ord 'a' ⇒ 97   toUpper 'a' ⇒ 'A'
chr 65 ⇒ 'A'   isDigit 'a' ⇒ False
```

Slide 2–30

Basic Types IV (a) – Tuples

- A Gofer tuple is similar to a Pascal record – it is a collection of objects of (a limited number of) objects, possibly of different types. Each Pascal record elements has a unique **name**, whereas in Gofer you distinguish between elements by their position in the tuple.
- Syntax: $(t_1, t_2, \cdots, t_n)$.

Examples:

```
type Complex = (Float,Float)
mkComplex ::  Float -> Float -> Complex
mkComplex re im = (re, im)
```

Slide 2–31

Basic Types IV (b) – Tuples

```
type Complex = (Float,Float)
mkComplex :: Float -> Float -> Complex
mkComplex re im = (re im)

mkComplex 5 3 ⇒ (5, 3)

addComplex :: Complex -> Complex -> Complex
addComplex (a,b) (c,d) = (a+c,b+d)

addComplex (mkComplex 5 3) (mkComplex 4 2) ⇒ (9,5)
```

Slide 2–32

Summary I

- B&W reading list:
  1. Fundamental concepts: 1–11
  2. Lists and strings: 48–50
  3. Basic data types: 16–18, 24–28
  4. Characters and strings: 28–30
  5. Tuples: 32–34
- Note that the language presented in B&W is not quite identical to Gofer. We'll describe the differences as we go.
- If you have a Mac or PC at home, you may consider downloading a copy of Gofer.
- The Gofer implementations are interactive and interpreted.

Slide 2–33

Summary II

- Gofer has all the basic types one might expect: `Ints`, `Chars`, `Floats`, and `Bools`.
- Gofer functions come in two parts, the signature and the declaration:

```
fun_name :: argument_types
fun_name param_names = fun_body
```

- Many Gofer functions will use recursion.
- Gofer doesn't have assignment statements, loop statements, or procedures.
- Gofer tuples are similar to records in other languages.

Slide 2–34

Homework I

1. Start (Mac `||` Unix) Gofer.
2. Enter the `commaint` function and try it out.
3. Enter the `addComplex` and `mkComplex` functions and try them out.
4. Turn on tracing (Options:Trace Reductions in MacGofer) and try the functions again.
5. Try the standard functions `fst x` and `snd x` on complex values. What do `fst` and `snd` do?
6. Try out the Eliza application in `Demos:Eliza`.

Slide 2–35

Homework II

- Write a Gofer function to check if a character is alphanumeric, i.e. a lower case letter, upper case letter, or digit.

Examples:

```
?  isAlphaNum 'a'
   True
?  isAlphaNum '1'
   True
?  isAlphaNum 'A'
   True
?  isAlphaNum ';'
   False
?  isAlphaNum '@'
   False
```

Slide 2–36

Homework III

- Define a Gofer exclusive-or function.

Template:

```
eOr ::  Bool -> Bool -> Bool
eOr x y = ···
```

Examples:

```
?  eOr True True
   False
?  eOr True False
   True
?  eOr False True
   True
?  eOr False False
   False
```

Slide 2–37

Homework IV

- Define a Gofer function `charToInt` which converts a digit like `'8'` to its integer value 8. The value of non-digits should be taken to be 0.

Template:

```
charToInt ::  Char -> Int
charToInt c = ···
```

Examples:

```
?  charToInt '8'
   8
?  charToInt '0'
   0
?  charToInt 'y'
   0
```

Slide 2–38