F.G.H.
5/16/85

LispCourse #28: Type Checking; Strings; Arrays

Type Checking in Interlisp

The Concept of Type Checking

Many programming languages insist that each parameter to a function be typed ÿ i.e., that it be declared to be a variable whose value is a given type of data (e.g., an litatom, a number, or a list).

In these languages when you call a function, the language checks the type of each argument in the argument list to make sure that it matches the declared type of each parameter in the parameter list.

An error results if you call a function with arguments of the wrong type.

Example from PASCAL:

```
function SumOfSquares (X: INTEGER, Y: INTEGER): INTEGER
BEGIN
    SumOfSquares := (X * X) + (Y * Y)
END
```

This PASCAL function definition says that *SumOfSquares* is a function that takes two integer arguments and returns an integer.

Therefore:

(SumOfSquares 2 3) is okay

But ***(SumOfSquares "Foo" 4)*** causes an immediate error because "Foo" is not an integer, but a string of characters.

The error will say something like "First argument to SumOfSquares is not an integer as required."

Note that ***(SumOfSquares 1.234 4)*** also causes an error because 1.234 is not an integer, but a floating point number.

Interlisp does not do this type checking. You do not have to declare the expected type of a parameter in your function definitions AND Interlisp does not check the type of the arguments when you call the function.

Example:

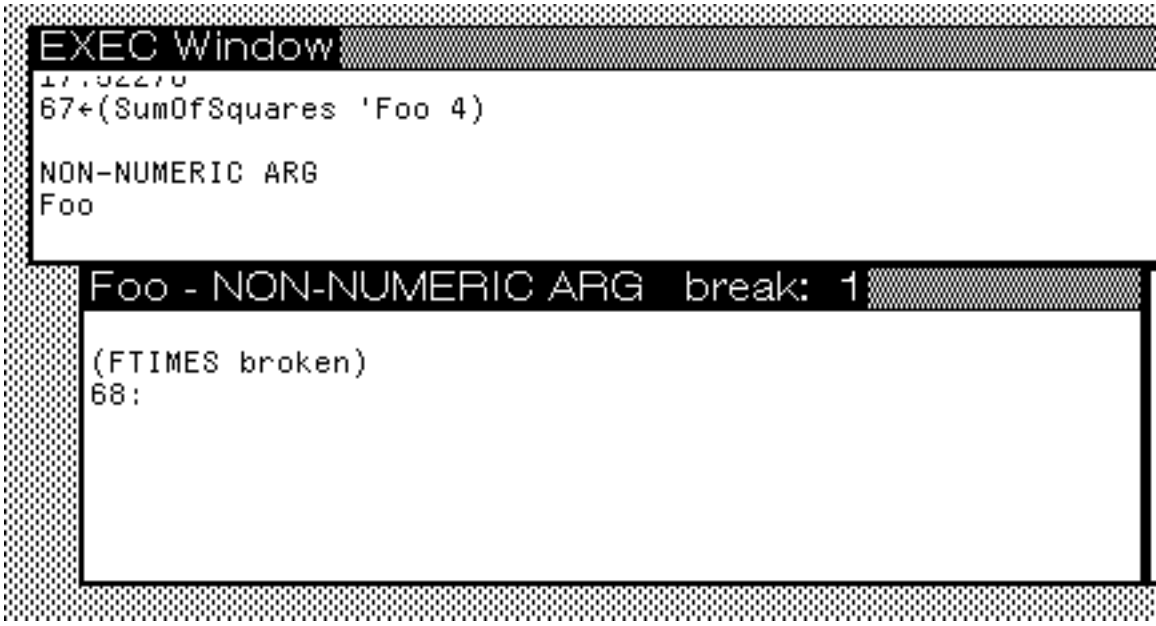
```
(DEFINEQ
    (SumOfSquares
        (LAMBDA (X Y)
            (PLUS (TIMES X X)(TIMES Y Y))))))
```

(SumOfSquares 2 3) works fine and returns an integer, 13.

(SumOfSquares 1.234 4) also works fine, but returns a floating point number 17.52276.

But, ***(SumOfSquares 'Foo 4)*** will cause an error (i.e., a BREAK).

The error will occur somewhere deep inside the SumOfSquares function and the error message will not reflect the real problem ž i.e., that the argument to SumOfSquares was of the wrong type.



Note that the error occurs in FTIMES, not in SumOfSquares where it really should have happened.

In this case, its easy to trace back to see that in fact SumOfSquares has a bad argument. But if these were complex functions embedded many levels deep, it could be very difficult to figure out where the ’Foo that tripped up the FTIMES actually came from.

This example points out both the strengths and weaknesses of the Interlisp scheme of no type checking.

The strength is that you can easily write a single function that handles many different kinds of data ÿ e.g., both integers and floating numbers.

The weakness is that data of the wrong type may be passed through many layers of a program before an error is tripped up. When this happens, it is very difficult to trace where the bad data came from.

Adding type checking to your Interlisp functions

The solution to Interlisp's lack of explicit type checking is to judiciously add type checking to the functions you write.

The basic goal of type checking is to catch data of the wrong type at the earliest point at which you can detect that it is wrong.

In particular, at any point where new data enters the system ž e.g., at user type-in or when reading from a database ž make sure the data is of the type expected.

For example, if you are getting information from a user to add to the database, check the arguments to the add-to-database function to make sure that the user is entering the correct type of data.

Also, at any other point where there is a possibilty of a wrong type of data being passed to a function, check the type of the arguments within that function.

It is important to type check only where necessary because type checking does take some time.

Too much type checking will make your program very inefficient.

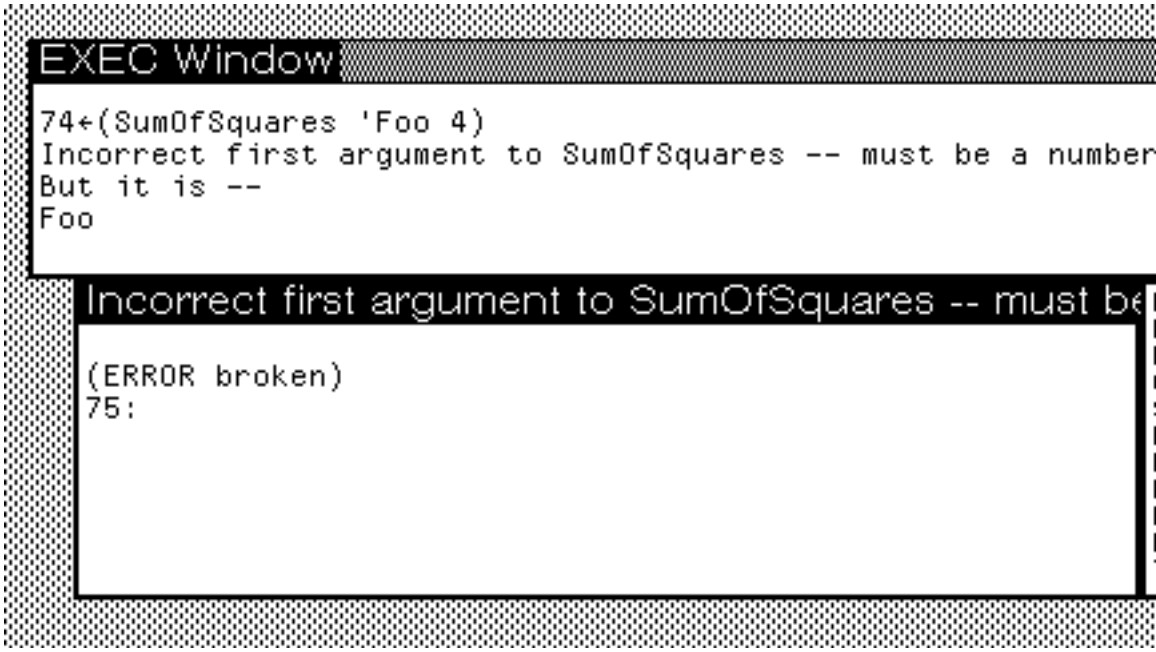
On the other hand, with too little type checking your program may not run at all.

Type Checking Technique

To check the arguments being passed to a function, just put a COND clause at the beginning of the function. The COND should have a cluase for each parameter that aborts the function and reports an error to the user if the value of the parameter is not an allowable type (i.e., if the calling function had an incorrectly typed argument).

Example:

```
(DEFINEQ
    (SumOfSquares (LAMBDA (X Y)
        (COND
            ((NOT (NUMBERP X))
                (ERROR "Incorrect first argument to
                SumOfSquares -- must be a number.  But it
                is --"  X))
            ((NOT (NUMBERP Y))
                (ERROR "Incorrect second argument to
                SumOfSquares -- must be a number.  But it
                is --")  Y)
            (T
                (PLUS (TIMES X X)(TIMES Y Y)))))))
```

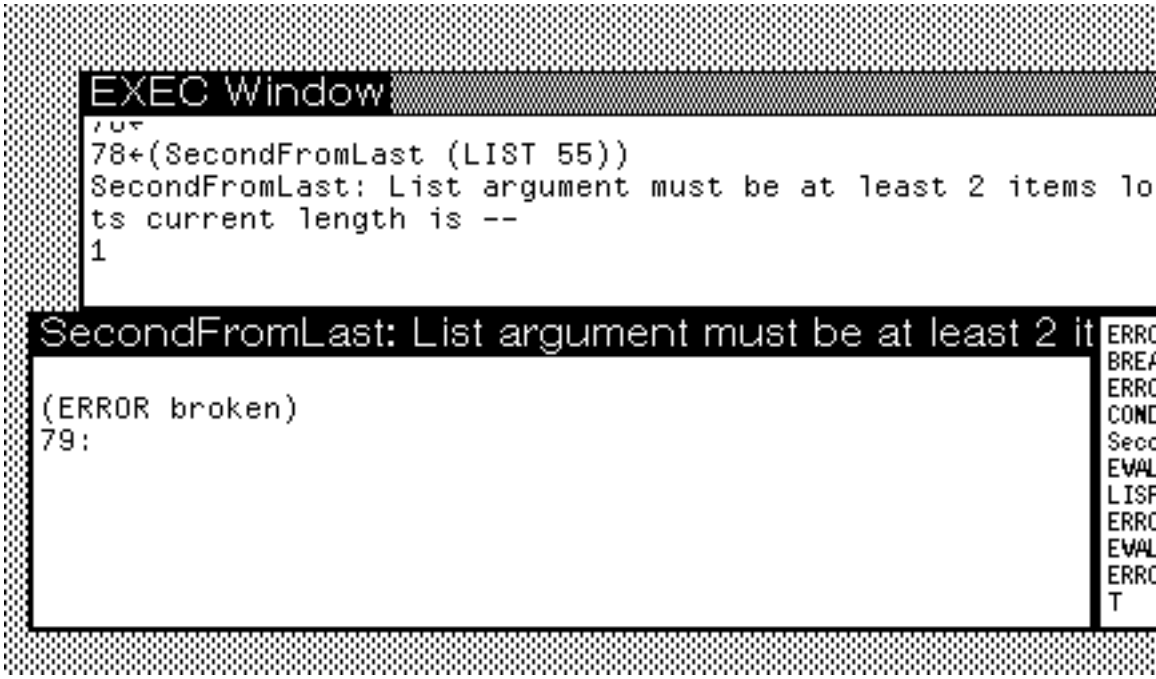


Second Example:

Define a function to return the tail of a list starting from its second to last element.

```
(DEFINEQ
    (SecondFromLast (LAMBDA (List)
        (* * Return the tail of a list starting from its second
        to last element.  Make sure List is in fact a list and
        is of length 2 or more.)

        (COND
            ((NOT (LISTP List))
                (ERROR "SecondFromLast:
                Argument must be a list.  It is --"
                List))
            ((LESSP (LENGTH List) 2)
                (ERROR "SecondFromLast: List
                argument must be at least 2 items
                long.  Its current length is --"
                (LENGTH List)))
            (T
                (NTH List (DIFFERENCE
                (LENGTH List) 2)))))))
```



Type checking predicates

The predicates necessary for checking the type of a given piece of data have already been discussed.

> LispLecture #4 (page 3), discusses the predicates **LITATOM**, **NUMBERP**, and **LISTP** that check if their argument is of type litatom, number, and list, respectively.
>
> LispLecture #25 (page 13) discusses the **TYPE?** statement that allows one to build predicates that check if their argument is a particular type of RECORD or a particular DATATYPE.

The next section contains an overview of the types of data in Interlisp, including the predicates that check for each of the types.

Overview of the Types of Data in Interlisp

Below is a list of the types of data supported by Interlisp. In parentheses following each data type is the predicate that checks for that data type.

Primitive data types (those provided as part of the core Interlisp system).

Atoms *(ATOM)*

Litatoms *(LITATOM)*

Numbers *(NUMBERP)*

Integers *(FIXP)*

Floating Point Numbers *(FLOATP)*

Lists *(LISTP)*

[Prop Lists]

[Assoc Lists]

Strings (STRINGPs)

Arrays (ARRAYPs)

Compound data types (those built by combining primitive data types)

RECORDs

RECORDs defined by the Interlisp implementors

SKETCHs *(TYPE? SKETCH)*

GRAPHs *(no type checking predicate!)*

...

User defined RECORDs *(TYPE? **RecordName**)*

DATATYPEs

DATATYPEs defined by the Interlisp implementors

BITMAPs *(BITMAPP)*

HARRAYs *(HARRAYP)*

WINDOWs *(WINDOWP)*

PROCESSes *(PROCESSP)*

STREAMs *(STREAMP)*

...

User defined DATATYPEs *(TYPE? **DataTypeName**)*

Print Names

Every data object in Interlisp has something called a ***print name*** (often called a ***pname***).

A data object's print name is the thing that gets printed when Lisp needs to communicate with the outside world ÿ e.g., in the PRINT phase of the READ-EVAL-PRINT loop.

Examples:

Atoms: the print name is the name of the atom. E.g., *FOOBAR*.

Lists: the print name starts with a "(", followed by the print names of all of the items in the list separated by spaces, followed by a ")". E.g., *(1 (2 3) A)*.

DATATYPES: the print name is the DatatypeName in "{ }" followed by some numbers. E.g., *{WINDOW}#65,12345* and *{PROCESS}#12,12399*

Print names are NOT unique.

For example:

{WINDOW}#65,1234 is the print name for some window. But it is also the print name for an atom whose name (i.e., print name) is exactly *{WINDOW}65,1234*.

For some data types, the Lisp object can be referred to by typing its print nameinto the Lisp Exec.

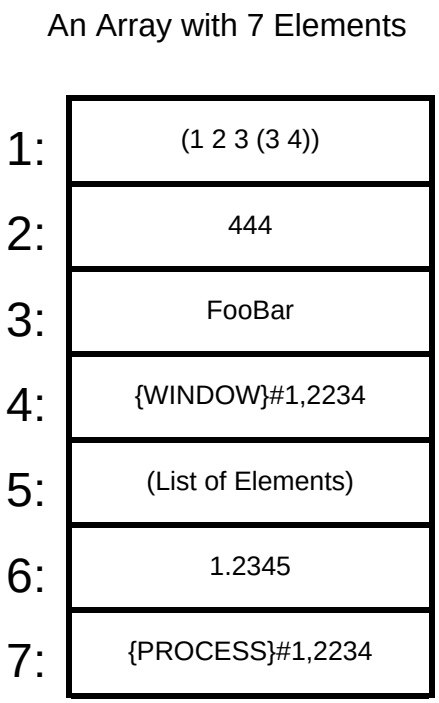
For example, to refer to an atom, you just type its print name (i.e., its name) into the Lisp Exec.

For other data types, the print name is just for printing; you can't type it back into the Lisp exec to refer to the object.

For example, if you type in *{WINDOW}#65,12345* into the Lisp Exec, the Lisp Exec will think you mean the atom by that name, not the window that has that pname.

Arrays

An array is a primitive data type that represents a fixed number (N) of other Interlisp objects stored in a one-dimensional vector.



Think of an array as a set of mailboxes arranged in one column and N rows. In each mailbox is some arbitrary Interlisp object (an atom, a list, a window, etc.) You can get at any object stored in the array of mailboxes only by specifying the row number of the mailbox it is stored in.

Alternatively, an array is like a RECORD with N fields, but the fields can be accessed by *number only* and *not by name*.

An array is also like a list, but it has a fixed length: you can't add or remove elements from an array.

Moreover, you can't deal with parts of the array as a single entity as you can a list (e.g., there is no operation like CDR for arrays).

For certain applications, arrays are much more efficient than lists.

In general, however, any program that uses arrays can be rewritten using lists ÿ with possible loss of efficiency and elegance.

Array manipulation functions

Interlisp has a number of functions that allow you to manipulate arrays, i.e., to create arrays, to access the objects stored in an array, etc.

Creating arrays

(ARRAY *Size*) ž Creates an array of size *Size* (i.e., with *Size* entries). Returns (a pointer to) the array. The array is initialized to have every element contain NIL.

Example:

1_(SETQ MyArray (ARRAY 10))
{ARRAY}#65,51054
2_MyArray
{ARRAY}#65,51054

Array predicate

(ARRAYP *Arg*) ž Returns *Arg* if *Arg* is an array, NIL otherwise.

Examples:

3_(ARRAYP MyArray)
{ARRAY}#65,51054
4_(ARRAYP 10)
NIL
5_ (ARRAYP (LIST 1 2 3 4 5))
NIL
6_ (ARRAYP 'ARRAY)

NIL

Accessing the entries of an array

(SETA *Array N Value*) ž Sets the *N*th element of *Array* to have the value *Value* (i.e., puts the Lisp object specified by *Value* into the *N*th element of *Array*).

Examples:

7_(SETA MyArray 1 (LIST 1 2 3))

(1 2 3)

8_(SETA MyArray 2 (PLUS 2 3))

5

9_ (SETA MyArray 12 15)

ILLEGAL ARG

12

10_ (SETA MyArray 3 15)

15

(ELT *Array N*) ž Returns the Lisp object stored in the *N*th element of *Array*.

Examples:

11_(ELT MyArray 1)

(1 2 3)

12_(ELT MyArray 2)

5

13_ (ELT MyArray 12)

ILLEGAL ARG

12

14_ (ELT MyArray 5)

NIL

Finding out the size of an array

(ARRAYSIZE *Array*) ž Returns the size of array *Array*.

Example:

15_(ARRAYSIZE MyArray)

10

16_ (ARRAYSIZE (ARRAY 50))

50

Using arrays

Problem:

Imagine you work for a company that has 20 products (numbered 1 thru 20). Each product has a "list price" and an "our price".

1. Write a function that takes a product number and returns the "list price".
2. Write a function that takes a product number and returns the "our price".
3. Write a function that replaces the price entry field with the atom OutOfStock for a given product number.

Solution:

Store the data in an array of size 20, where each entry is a RECORD called *Prices* with 2 fields named *ListPrice* and *OurPrice*.

```
(SETQ PriceArray (ARRAY 20))

(SETA PriceArray 1 (CREATE Prices ListPrice _ 1.00 OurPrice 1.25))

... {Fill in rest of proce array with values}

(DEFINEQ
        (LC.ListPrice (LAMBDA (ProductNumber)
```

```
                    (fetch (Prices ListPrice) of
                        (ELT PriceArray ProductNumber)))
                (LC.OurPrice (LAMBDA (ProductNumber)
                    (fetch (Prices OurPrice) of
                        (ELT PriceArray ProductNumber)))
                (LC.MarkOutOfStock (LAMBDA (ProductNumber)
                    (SETA PriceArray ProductNumber 'OutOfStock))))
```

The advantage of using an array in this case is that you need to get to and CHANGE any element of the data structure at any time.

This is easy with arrays using ELT and SETA.

It is harder with lists.

First, (CAR (NTH List N)) takes longer than (ELT Array N).

Second, there is no easy way to do SETA with list structures. (Though we will learn how to do so not easily later!!!).

Strings

Strings are a primitive data type in Interlisp used for representing sequences of characters. (As opposed to atoms which are used as symbols for arbitrary objects.)

A string is an arbitrary sequence of characters, including spaces and tabs.

The print name of a string encloses the characters in the string in double quotes.

Examples:

"B"

"abc"

"Frank G. Halasz"

"This is a very long string. It consists of several sentences. The sentences are separated by spaces."

Strings can be from 0 to any number of characters in length.

The string "" is the empty string having 0 characters.

Strings can contain any characters except the double quote character and %.

To include these characters they must be preceded by the % escape as in atom names.

Example:

"String with single %% percent sign"

String manipulation functions

Interlisp has a number of functions that allow you to manipulate strings, i.e., to create strings, to concatenate strings, to decompose strings, to search through strings, etc.

String predicate

(STRINGP *Arg*) ž Returns *Arg* if *Arg* is a string, NIL otherwise.

Examples:

1_(STRINGP "ABC")

"ABC"

2_(STRINGP 'ABCDEF)

NIL

3_(STRINGP (LIST 1 2 3 4))

NIL

8_(STRINGP (TEDIT))

NIL

Creating strings

(MKSTRING *Arg*) ž If *Arg* is already a string, returns *Arg*. Otherwise, makes and returns a string containing the print name of *Arg*.

Examples:

5_(MKSTRING "ABC")

"ABC"

6_(MKSTRING 'ABCDEF)

"ABCDEF"

7_(MKSTRING (LIST 1 2 3 4))

"(1 2 3 4)"

8_(MKSTRING (TEDIT))

"{PROCESS}#61,130000"

(ALLOCSTRING *N Character*) ž Returns a string *N* characters long where each character is *Character*. *Character* can be a single character string/atom or a character code (see LispCourse #10, page 5)

Examples:

9_(ALLOCSTRING 5 "A")

"AAAAA"

10_(ALLOCSTRING 15 'B)

"BBBBBBBBBBBBBBB"

11_(ALLOCSTRING (PLUS 3 4) 63)

"???????"

12_(ALLOCSTRING 7 (CHARCODE ?))
"???????"

Comparing strings

(STREQUAL *Str1 Str2*) ž Returns T is *Str1* and *Str2* are both strings and contain the same sequence of characters.

Examples:

13_(STREQUAL "ABCDEF "ABCDEF")
T
14_(STREQUAL (ALLOCSTRING 5 ’A) "AAAAA")
T
15_(STREQUAL ’A "A")
NIL
16_(STREQUAL (MKSTRING ’AAA)(ALLOCSTRING 3 "A"))
T

Concatenating strings

(CONCAT *Str1 Str2* ...) ž Returns a new string that consists of the concatenation of the characters in *Str1, Str2, Str3* .... If any *StrI* is not a string, the MKSTRING of that *STRI* is used instead of *StrI*.

Examples:

17_(CONCAT "ABCDEF "GHIJKL")
"ABCDEFGHIJKL"
18_(CONCAT (ALLOCSTRING 5 ’A) "FOO BAR")
"AAAAAFOO BAR"
19_(CONCAT 1234 " " 5678 " " (LIST 9 0))
"1234 5678 (9 0)"
20_(CONCAT "This is the kind of value that TEdit returns -- " (TEDIT))

"This is the kind of value that TEdit returns -- {PROCESS}#61,130000"

Decomposing strings

(SUBSTRING *Str Start End*) ž Returns a new string that consists of the characters of *Str* starting at character number *Start* and ending at character number *End*.

If *End* is NIL, the last character of *Str* is used.

If *Start* or *End* are negative, they are interpreted as being positions from the end of *Str*.

Examples:

21_(SUBSTRING "ABCDEF 2 4)

"BCD"

22_(SUBSTRING "FOO BAR" 4)

" BAR"

23_(SUBSTRING "FOO BAR" ÿ3)

"BAR"

24_(SUBSTRING "FOO BAR" 2 -2)

"OO BA"

Searching strings

(STRPOS *Pattern String Start SkipChar*) ž Searches through string *String* looking for any sequences of characters that matches the characters in string *Pattern*. If a match is found, STRPOS returns the number of the character in *String* where the match starts. If no match is found, STRPOS returns NIL.

If *Start* is specified, the search begins at character number *Start* in *String*.

If *SkipChar* is specified, any instance of *SkipChar* in the *Pattern* string will match any character in *String*. (*SkipChar* is the wildcard character).

If *Pattern* and/or *String* are not strings, their MKSTRINGs will be used instead.

Examples:

```
25_(STRPOS "Q" "ABCDEF")
NIL
26_(STRPOS "D" "ABCDEF")
4
27_(STRPOS "C*E" "ABCDEF" NIL "*")
3
28_(STRPOS "O" "FOO BAR" 4)
NIL
```

Using Strings

Given a list of strings of the format: "Name: *Last,First*".

Write a function to extract all names with "sz" in them. The function should return a list of strings with the format "*First Last*".

```
(DEFINEQ
    (LC.szP (LAMBDA (String)
        (* * Does String have an sz in it?)
            (OR (STRPOS "sz" String)
                (STRPOS "Sz" String)
                (STRPOS "sZ" String)
                (STRPOS "SZ" String))))

    (LC.GetLastName (LAMBDA (String)
        (* * Extract the last name from the string)
            (SUBSTRING String
                (PLUS 1 (STRPOS " " String))
                (DIFFERENCE (STRPOS "," String) 1))))

    (LC.GetFirstName (LAMBDA (String)
```

```
        (* * Extract the first name from the string)
            (SUBSTRING String (PLUS 1 (STRPOS "," String)))))

(LC.FindSzNames (LAMBDA (List)
        (* * Extract all names with sz in the last name)
            (FOR Entry in List
                WHEN (LC.szP Entry)
                COLLECT
                    (CONCAT
                        (LC.GetFirstName Entry)
                        " "
                        (LC.GetLastName Entry))))))

6_(SETQ TestList (QUOTE
                ("Name: Halasz,Frank" "Name: Smith, Sam"
                "Name: Beals, Szmatha"  "Name: Schatz,Sheila")))
("Name: Halasz,Frank" "Name: Smith, Sam" "Name: Beals, Szmatha"
"Name: Schatz,Sheila")
7_(LC.szP (CAR TestList))
11
8_(LC.GetFirstName (CAR TestList))
"Frank"
9_(LC.GetLastName (CAR TestList))
"Halasz"
10_(LC.FindSzNames TestList)
("Frank Halasz" " Szmatha Beals")
```

References

In general, primitive data types are covered in Chapter 2 of the IRM.

Arrays are covered in Section 2.7.

Strings are covered in Section 2.6.