AI PROGRAMMING II
Intelligent Systems MSc
Benedict du Boulay

These notes are copied from those
for 1998 written by Rudi Lutz and adapted
by Chris Taylor.

The course for 2001-02 will vary
in some details.

December 24, 2001

IS MSc
AI Programming II

Topic 0

Introduction to Course

Overview of Lecture

- Course outline
  - this term in detail
- General information
  - teaching times
  - getting help
  - on-line documentation
- Getting started
  - Documentation
  - Where to go from there

Course Outline

- See Handout
- Purpose of course
  - Familiarise with AI languages and programming
  - Familiarise with procedural languages and programming
  - Familiarise with process of program development and documentation

Tutor Ben du Boulay
Office 3R346
email bend

Content

- This term:
  - Pop-11 in POPLOG programming environment
  - also some Lisp
  - Working knowledge of **Unix** operating system
  - Working knowledge of **VED** (a visual/screen editor) - like a simple word processor with extra features to support programming.
  - See handout for details of lectures
- Last term:
  - Prolog (logic-based AI language)

Assessment

- Project handed in after Easter
- mark combined with similar project in Prolog handed in after Christmas

General Information

- Lectures
  - Two per week. See handout for full details.
  - Week 9 — Lisp
- Assignments
  - Set exercises to be handed in the following week at exercise classes.

Teaching Times

- Lectures will be the following times (starting Week 1):

Wednesdays	12.30–13.20	Arundel 404A
Thursdays	17.00–17.50	Pevensey 1 2A3

- Exercise Class once per week (Pevensey 1 2D12), Thursdays 11.30–12.20

Other Sources of Help

- Demonstrator in labs (times to be announced)
- Each other!!
- On-line documentation
  - Teach files
  - Help files
  - Ref files (advanced)
  - Web
- Books (see handout)
  - The Pop-11 Primer (TEACH PRIMER)
  - VED User Guide
  - Photocopies of these lecture slides
  - Buy these in 4C18.

- Once you get going online
    - Teach files take over
    - Try

```
TEACH TEACH
TEACH VED
⋮
TEACH RESPOND
```

Getting Into POPLOG - three ways

- From the **Unix prompt** % .

  tsunx-% *teach teach <RETURN>*

  or

  tsunx-% *teach ved <RETURN>* takes you into POPLOG via VED

- From the CDE Desktop.

  Left click on tab above XE button

  Left click on XVED

  Press the <ENTER> key

  Type: *teach teach* or *teach ved*

- From the CDE Desktop via Applications.

  Left click on tab above Apps. button

  Left click on Applications

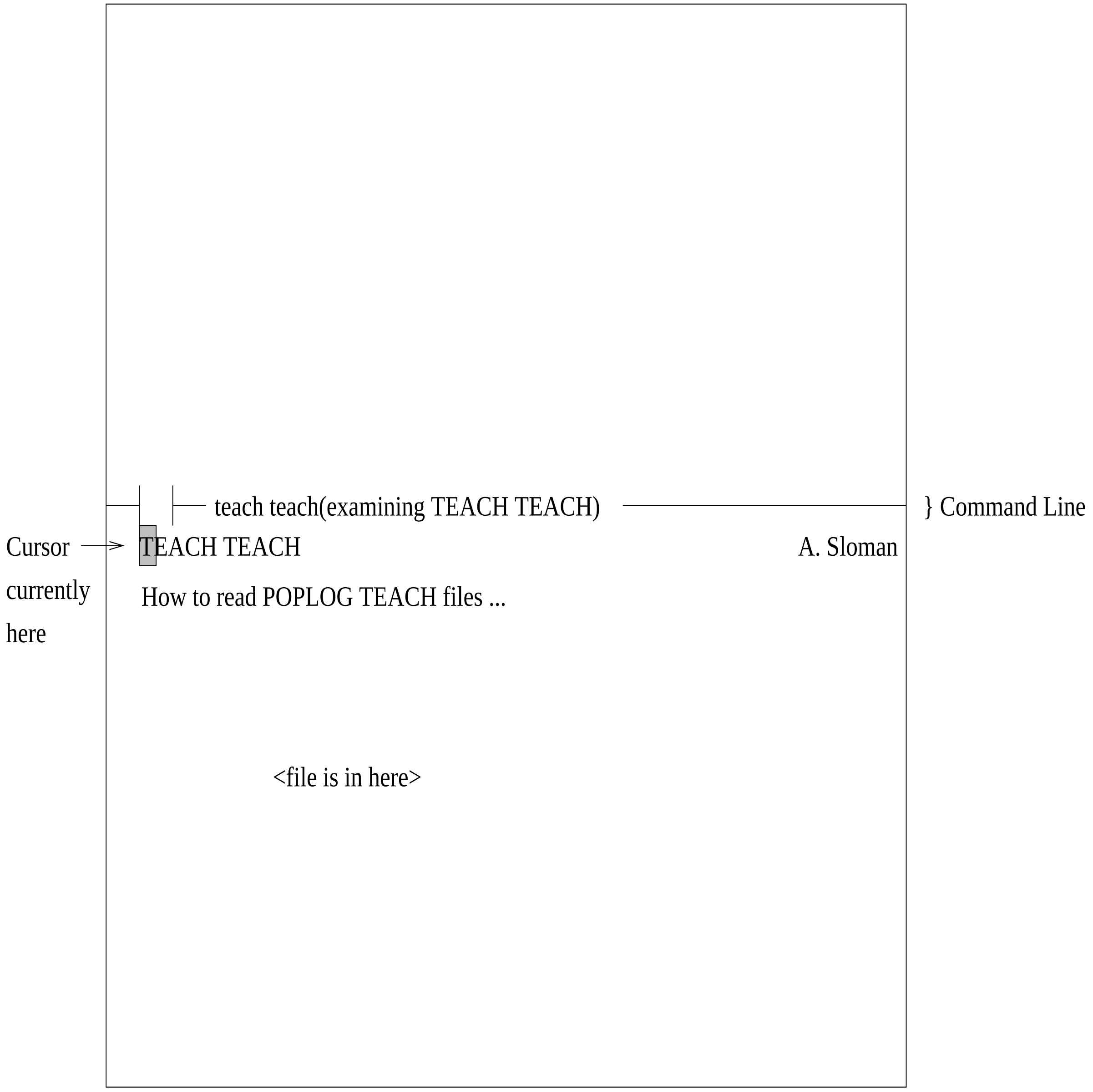
  Double left click on Poplog_Apps

  Double left click on XVED

  Press the <ENTER> key

  Type: *teach teach* or *teach ved*

Once inside POPLOG



teach teach(examining TEACH TEACH)

} Command Line

Cursor currently here

TEACH TEACH

A. Sloman

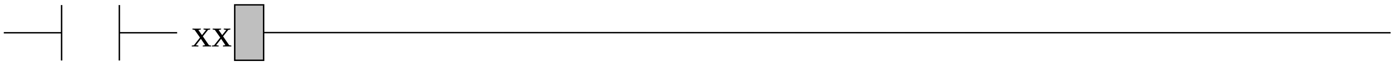
How to read POPLOG TEACH files ...

<file is in here>

Getting out of Ved, and logging out

Press <ENTER> key, moves cursor to command line

Can now type commands e.g. xx



If press <RETURN> command will be executed.

xx quits POPLOG, and takes you back to UNIX(shell).

or close window by left clicking on close window icon

```
define doctor();
    lvars answer;
    [are you feeling well] =>
    readline() -> answer;
    if answer = [yes] then [you do not need me] =>
    else feelbad()
    endif;
    [that will be $50 please] =>
enddefine;
define feelbad();
    lvars answer;
    [do you hurt somewhere] =>
    readline() -> answer;
    if answer = [yes] then [take two aspirins] =>
    else [you need to see a specialist] =>
    endif
enddefine;
```

```
;;; DECLARING VARIABLE feelbad
doctor();
** [are you feeling well]
? no
** [do you hurt somewhere]
? yes
** [take two aspirins]
** [that will be $ 50 please]

doctor();
** [are you feeling well]
? yes
** [you do not need me]
** [that will be $ 50 please]
```

Running Pop-11 Code

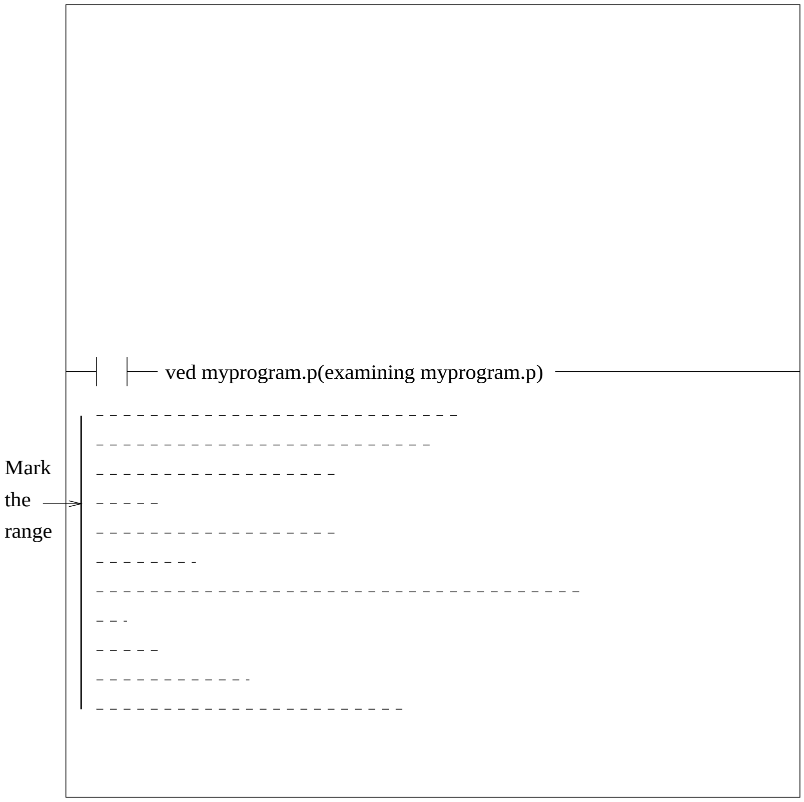
- Calling Pop-11 from VED (See TEACH VEDPOP)
  - usual mode of access
  - enables one to keep copies of programs
  - enables making of quick changes
- To edit an existing file, or to create a new one

  tsunx% *ved myprogram.p* (to Unix)

  OR

  <*ENTER*>*ved myprogram.p* (to VED)

  Note ".p" suffix to file name indicates that the file contains Pop-11 code.



- Either <ENTER>lmr (i.e. load marked range),
  OR

  simply <CTRL>d

  will cause Pop-11 to “execute” the marked code.

Compiling/Running Lines of Code

- from XVED
  - Mark a range using **f1**, **f2** keys, type <ENTER> lmr *or* <CTRL>d
  - Use *compile* menu button and select line, range, procedure, file
  - <ENTER> l1
- From POP-11 prompt
  - load <*filename.p*>; e.g. load surgery.p;
  - compile('surgery.p');
- Within a program
  - compile(' bend/mypoject/surgery.p');

Contents of file dissertation.p

- Makes use of three libraries of pre-defined procedures e.g. for a production system or a semantic network and four files of your own procedures. Each file contains one or more related procedures.

-
  ```
  uses library1, library2, library3;
  load myfile1.p;
  load myfile2.p;
  load myfile3.p;
  load myfile4.p;
  ```

IS MSc
AI Programming II

Topic 1

Introduction to Computing

Introduction to Computing

- VED
  - What it looks like
  - VED facilities
  - access from VED
- Unix
  - Calling Unix from VED
  - Unix facilities
- Pop-11
  - Calling Pop-11 from VED
  - Loading programs
  - Running programs
  - Output file

VED (See VED User Guide)

- **V**isual display(screen) **ED**itor
  - Interfaces to Unix(shell)
  - Interfaces to programming environments: Pop-11, Lisp, Prolog, ML.
    See TEACH VEDPOP
- What happens when you call VED?
  - Puts a “file” on the screen
  - **Copied** from disk to VED “buffer”
  - Writing to disk
    - Backup copies
    - Quotas/filespace

What VED looks like

- Split screen (on some terminals multiple “windows” instead
- Command Line (+ <ENTER>)
- Copy of file containing text and/or programs(code)
- Two files can be displayed at any one time (usual mode)
- Can enlarge one file to full screen/window by <ESC>w
- To reduce again, repeat <ESC>w
- Other files “hidden”
  - Names kept in ved bufferlist
  - To see it, type <ESC>e then select filename

VED Facilities

- Facilities for moving around files
- For moving between files — <ESC>x
- For modifying contents
- Static mode — <ENTER>static
- Writing contents
  - <ENTER>w
  - <ENTER>wq
  - <ENTER>qq
  - <ENTER>xx
  - <ESC>q
- Marking ranges
- Deleting ranges — <ENTER>d

- Moving/copying marked ranges
  - Within file — <ENTER>m and <ENTER>t
  - Between files — <ENTER>mi <ENTER>mo <ENTER>ti <ENTER>to
- “Loading” marked ranges — <ENTER>lmr
  - See TEACH VEDPOP
- Searching a file — <ENTER>/*wanted text*
- Global edits — <ENTER>s/*string*/*replacement*
- Redo
- The command line is a ved buffer — can move up and down it using normal keys

Access from VED

- To other files in VED bufferlist
  - <ESC>x
  - <ESC>e
- To other files
  - <ENTER>ved *filename* will
    - will copy file from disk into new ved buffer (if file not in ved bufferlist)
    - display existing buffer if file in ved bufferlist
    - create a new buffer(with no counterpart on disk yet) if file does not exist

- To Unix
  - <ENTER>stop
    will return you to shell, without terminating VED. Type % at the Unix prompt (i.e. you *actually type* a percent sign as a Unix command!) to return to VED.
  - <ENTER>% is similar, but you are in a **new shell**. Type <CTRL>d to get back
    BEWARE!!!
  - <ENTER>%*command*
    e.g. <ENTER>%ls
- To Pop-11
  - <ENTER>lmr — load marked range
  - <ENTER>l1 load current file only

Warning

In case of either <ENTER>stop or
<ENTER>%

DO NOT INVOKE VED AGAIN INSIDE THE SHELL.

It will get a **new copy** of the file from disk. Beware of losing work

For most purposes use
<ENTER>%*command*

IS MSc
AI Programming II

Topic 2

Objects and Expressions

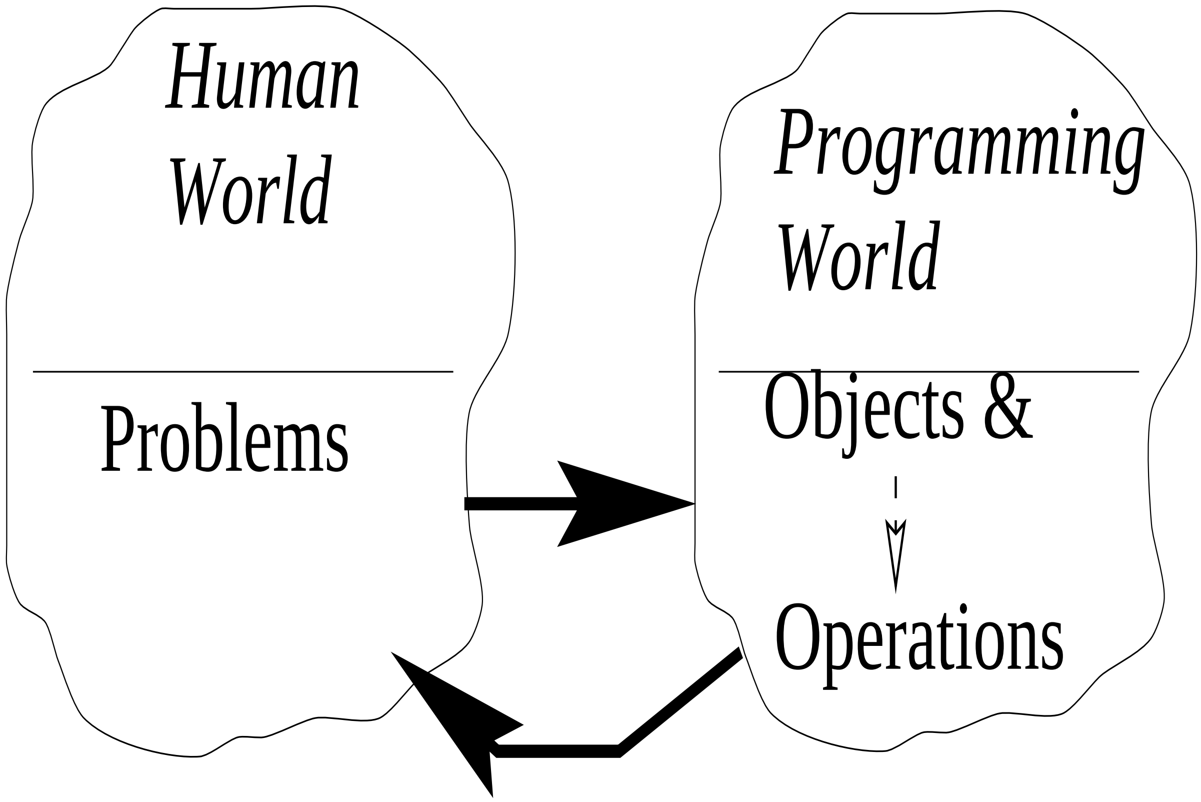
Pop-11 Programming

- This course will introduce
  - An AI programming language
  - Programming techniques and methodology
  - “Behind the scenes” explanations
- Will begin with Objects
  - Different types (data types)
  - primitive actions on these

The programming process

- Programming consists of
  1) Choosing a representation for the relevant features of the problem in terms of data objects provided by the programming language
  2) Finding a suitable sequence of operations acting on these data objects which compute a data object (or objects) which when interpreted in terms of the problem (as defined by step 1) gives one a solution to the original problem.

Programming Process



Human World

Problems

Programming World

Objects &

Operations

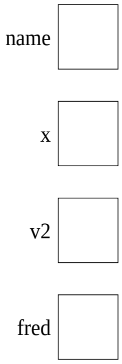
Different programming languages provide different objects and operations

Data Types

- **numbers**
  - **integers** e.g. ...,-3,-2,-1,0,1,2,3...
  - **decimals** e.g. 1.359, 2.6, 1.0, -3.56, etc.
- **words** e.g. "rudi" "cat" "computer" "foo" "baz39"
- **strings** e.g. 'My name is Rudi' 'cat'
- **lists** e.g.
  - [a b c 5 6 d e]
  - [the black cat]
  - [1 2 3 4]
  - [ ] (the empty list)
- **booleans** — only two values are <true> and <false>
- Lots of others!

The Virtual Machine(Introduction)

- Provides
  - Variables
  - User stack
  - The heap
  - other things as well
- Variables
  - Think of as **named boxes** that can hold(contain/store) data objects



name $\boxed{3}$ x $\boxed{\text{undef}}$ v2 $\boxed{97}$ fred $\boxed{5}$

- Created by **variable declarations**

e.g. vars name;
vars x, v2;
vars fred;

Some Simple Actions

- The “print arrow” =>

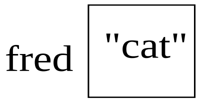
e.g. ”fred”=>

```
    ** fred       (this is what is printed)

    3=>
    ** 3

    [a b c]=>
    ** [a b c]
```

- The “assignment arrow” –>

e.g. ”cat” –>fred;

stores the **word** ”cat” in **variable** fred



fred $\boxed{\text{”cat”}}$

```
fred=>
** cat
```

Arithmetic Operations

- Most are “infix” operations

e.g.

```
2+3=>
** 5


6-12=>
** -6
```

- How does Pop-11 “know” that

  ```
  3+6*4
  ```

  evaluates to 27 (**not** 36)
- Each arithmetic operator (+ - * / etc) has a precedence associated with it.
- Operations with the lowest precedence are done first
- So $*$ and / are done before + and -

- Operators with same precedence are done from left to right
- If we want to override the normal precedence we **use parentheses**. Expression in parentheses(brackets) are evaluated first.

e.g. (3+6)*4=>

** 36

Running Pop-11 Code

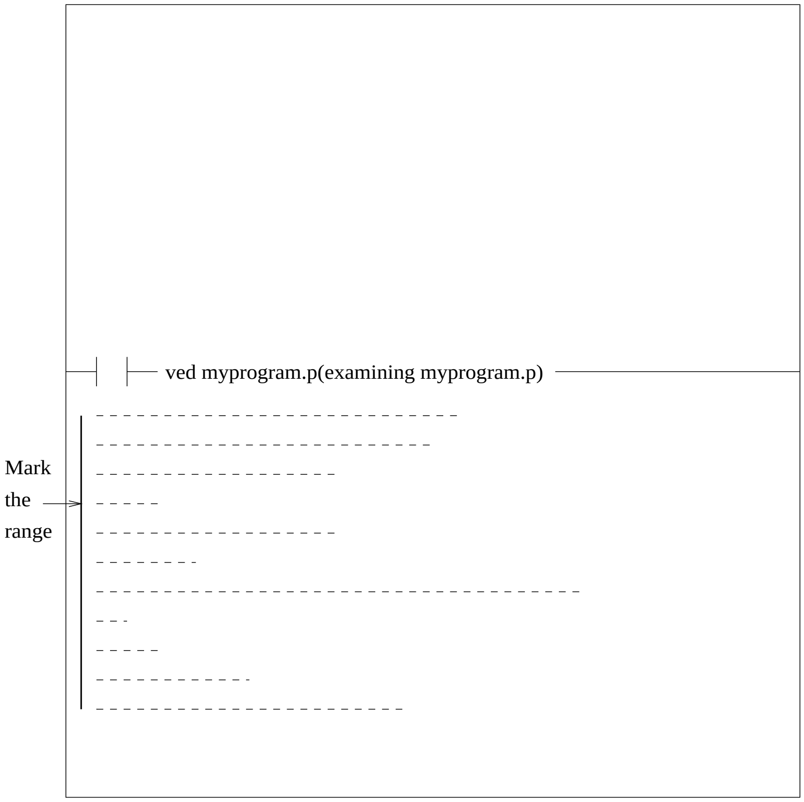
- Calling Pop-11 from VED (See TEACH VEDPOP)
  - usual mode of access
  - enables one to keep copies of programs
  - enables making of quick changes
- To edit an existing file, or to create a new one

  tsunb% *ved myprogram.p* (to Unix)

  OR

  <ENTER>ved myprogram.p (to VED)

  Note ".p" suffix to file name indicates that the file contains Pop-11 code.

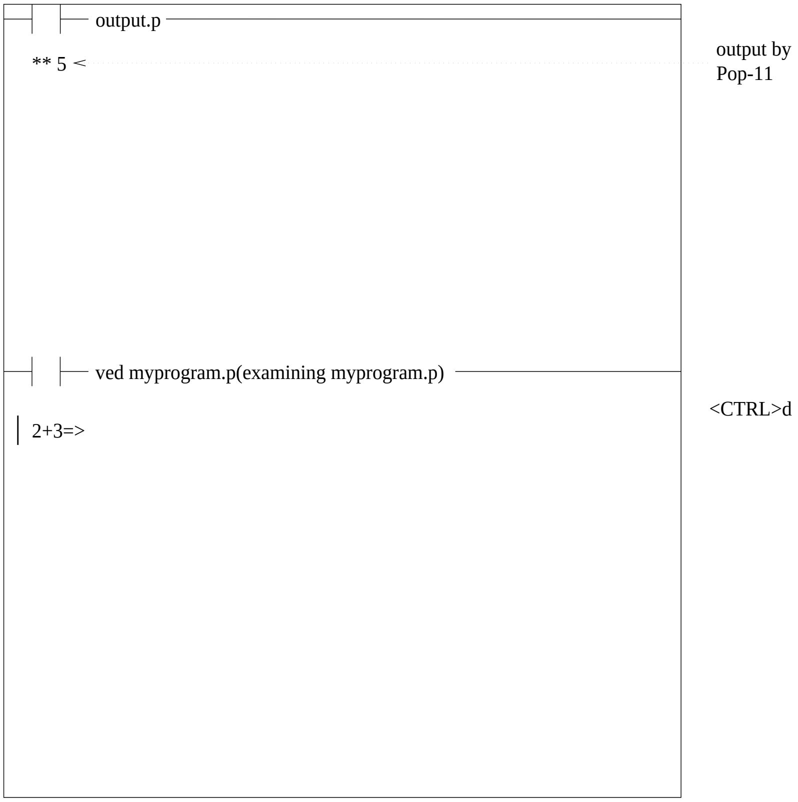


- Either <ENTER>lmr (i.e. load marked range),
  OR

  simply <CTRL>d

  will cause Pop-11 to “execute” the marked code.

Output from running programs



```
** 5 <
```

```
2+3=>
```

If executing the code involves any output (e.g. printing) this will be put in a file (actually the corresponding VED buffer) called **output.p**

List Operations

- The “head” operation **hd** obtains the first element of a list

e.g.
```
hd([a b c])=>
** a
```

- The “tail” operation **tl** obtains the list “minus” its first element

e.g.
```
tl([a b c])=>
** [b c]
```

```
vars mylist;
[the black cat]->mylist;
hd(mylist)=>

** the

tl(mylist)=>
** [black cat]
```

- Exercises (do these mentally, or on paper, first, then try them out on the machine to check your answer

  hd([1 2 3])=>

  tl([1 2 3])=>

  hd(tl([1 2 3]))=>

  hd(hd([1 2 3]))=>

  hd([ ])=>

  tl([a])=>

  tl([ ])=>

Stacks

- A stack is like pile of plates

———
———
———
———
———
———

- Plates can be added to the **top** of the stack.
- Plates can be removed from the **top** of the stack.
- Last In First Out (LIFO)
- **Jargon**

  push an object **onto** the top of the stack

  pop an object **off** the stack.

The User Stack

- Pop-11 has a stack called the **user stack**, onto which data objects are pushed, and off which they are popped
- The user stack (usually referred to as simply **the stack**) takes part in almost everything that happens in Pop-11
- It is probably one of Pop-11's most controversial features
- The use of a data object or variable name anywhere in Pop-11 *except to the right of an assignment* means **"put the object OR value of the variable on the stack"**

Examples

```
3;
```

results in

```
3
```

while

```
vars fred;
5->fred;
fred;
```

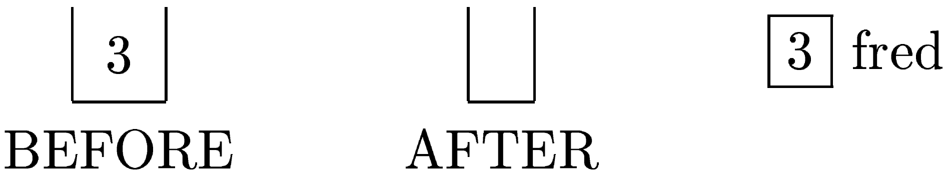
results in

```
5
```

Assignment and the stack

- The assignment arrow means **"take the top object off the stack and put it in the "box" named on the right of the assignment arrow"**
- So, **3–>fred** really means

3	–>fred
put 3 on the stack	take top object off the stack and store in fred



The stack and arithmetic operations

- **2+3** really means **2;3;ADD**

Put 2 on the stack	2
Put 3 on the stack	3 2
Take the top two items off the stack, add together and put result on stack	5

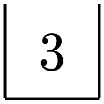
- Suppose we have done

```
vars x, y, z;
3->x;
4->y;
```

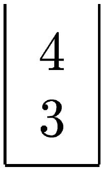
giving us $x\boxed{3}$ $y\boxed{4}$

If we then do **x+y->z;**

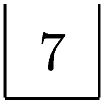
put **value** of x on stack



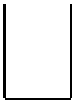
put **value** of y on stack



Add



take top item off stack
and store in z



$z\boxed{7}$

- **Note** x and y have not changed

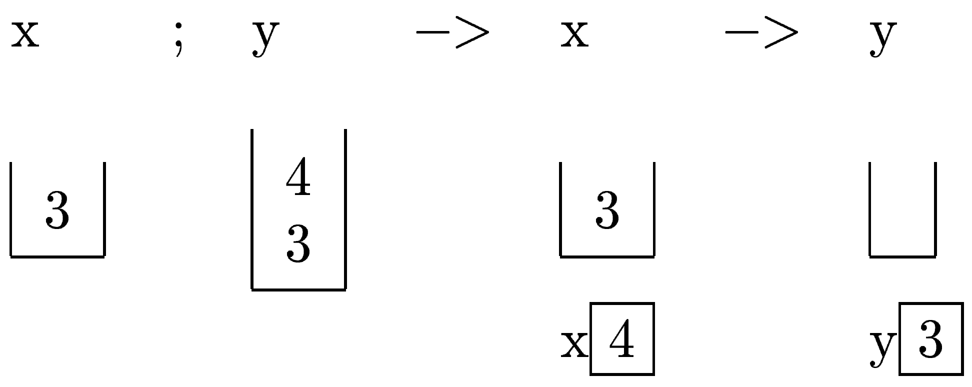
they are still $x\boxed{3}$ $y\boxed{4}$

Swapping the values of variables

Suppose we have x$\boxed{3}$ y$\boxed{4}$

Then **x;y−>x−>y;**

results in x$\boxed{4}$ y$\boxed{3}$



More than one result

- Some operations can leave more than one thing on the stack
- For example **dest** splits a list, putting the head on the stack, followed by the tail

```
dest([a b c])->list->x;
```

gives

```
list=>
** [b c]

x=>
** a
```

List Building Operations

- The <> operator can be used to join lists

e.g.

```
[a]<>[b c]<>[d e]=>
** [a b c d e]
```

e.g.

```
vars list1 list2 list3;
[a b c]->list1;
[d e f]->list2;
list1<>list2->list3;

list1=>
** [a b c]

list2=>
** [d e f]

list3=>
** [a b c d e f]
```

- The **::** operator can be used to add a new element to the front of a list

e.g.

```
[a b c]->list;
93::list->list1;

list=>
** [a b c]

list1=>
** [93 a b c]
```

Computing Elements of Lists

- Elements in a list are not normally evaluated

e.g.
```
[2+3]=>
** [2 + 3]

[name address]=>
** [name address]
```

- if we want to evaluate elements in a list we can use ˆ and ˆˆ

e.g.
```
"rudi"->name;
[15 Hendon Street]->address;

[ˆname ˆaddress]=>
** [rudi [15 Hendon Street]]

[ˆname ˆˆaddress]=>
** [rudi 15 Hendon Street]
```

- **ˆˆcan only be used on lists**

More List Accessing Operations

```
vars names;
[rudi sue ruth linda john]->names;

names(1)=>
** rudi

names(4)=>
** linda

[[jane sue] [tim mark]]->names;

names(2)(1)=>
** tim
```

Modifying (Updating) Lists

- Lists can be modified(updated) in various ways

e.g.
```
[a b c]–>list;

list=>
** [a b c]

4–>hd(list);

list=>
** [4 b c]

7–>list(3);

list=>
** [4 b 7]

[10 11 12]–>tl(list);

list=>
** [4 10 11 12]
```

- **Use updating operations WITH CARE!!!**

Vectors

- Vectors are **fixed length** structures similar in some ways to lists

e.g. {a b c} is a 3 element vector

- Cannot use hd and tl. Access by **indexing**

e.g. {a b c}−>vec;

```
vec(2)=>
** b

7->vec(2);

vec=>
** {a 7 c}
```

- Can use ^ and ^^ in vectors
- <> will join vectors
- In fact <> can be used to join words, lists, strings, vectors
- Strings are a restricted form of vector

Testing Objects for Equality

- There are two equality tests (both infix) available in Pop-11

  = tests for "similarity"

  == tests for "identity"

e.g.

```
[a b c]->list1;
[a b c]->list2;
list2->list3;

list1=list2=>
** <true>

list1==list2=>
** <false>

list2=list3=>
** <true>

list2==list3=>
** <true>
```

since list1 and list2 are **different lists** with **the same components** while list2 and list3 **are the same list**

- Any two objects which are == are also =
- Any two objects which are = are not usually ==
- Exceptions
    1. Two **words** which are = are also **always** ==
       e.g. 'cat"=="cat"=>
       ** <true>
    2. Similarly for (small) integers
       e.g. 3==3=>
       ** <true>
- Lists and other compound data types do not have this property

e.g. [a b c]==[a b c] =>
** <false>

- A deeper explanation will be given later, but if you are curious ask in tutorials/exercise classes

IS MSc
AI Programming II

Topic 3

Procedures

Procedures

- A **procedure** is a “packaged” sequence of Pop-11 statements
- Several uses
  - When you want to perform a set of statements more than once in a program
  - When you want to apply the same operations to different sets of data
  - To improve program readability

Defining A Procedure

- Procedures usually begin with the word **define** and end with a corresponding **enddefine**

e.g.
```
define greet();
    [Hello how are you today]=>
    [Enjoy your programming session]=>
    [Just type bye to finish]=>
enddefine;
```

- The line beginning **define** is known as the procedure **header**
- The code between the **define** and the **enddefine** is known as the procedure **body**
- A piece of code of the form

  `greet()`

  is known as a **call** of the procedure

```
define simple(num1,num2);
   num1+num2=>
enddefine;
```

- num1 and num2 are known as simple's **input local variables**, or more simply, its **input locals**.
- They are also known as its **formal parameters**
- simple can be **called** with different data

```
e.g. simple(4,6);
     ** 10
e.g. simple(7,9);
     ** 16
```

- The values given to simple in a call are known as its **actual parameters**

- When a procedure is called the values of its actual parameters are “passed into” its formal parameters *before* the body of the procedure is executed
- When a procedure finishes its execution the program continues executing any code that occurs(textually) after the call. The procedure is said to **return** to where it was called from
- hd, tl, +, -, $*$, /, <>, ::, are all examples of **built in procedures**

Local Variables of a Procedure

- Variables declared by a **lvars** statement **inside a procedure**, always have a **new** “box” created for them every time the procedure is called
- Once created this “box” will then exist until the procedure is exited. **Aside:** *a slight white lie - very occasionally they can last longer than this*
- Such a variable(named box) can only be accessed **from inside the procedure**
- This named “box” is independent of any other “boxes” with the same name that may exist elsewhere in the program
- Such variables are called **(lexical) local variables** of the procedure.

An Example

```
vars x;
define simple();
lvars x;
   27->x;
   x=>
enddefine;

100->x;
simple();
** 27

x=>
** 100
```

- So, if you need extra variables to hold intermediate results inside a procedure, declare them local to the procedure, but do not worry about “name clashes”

Procedure Results and Outputs

- Usually don't just want the results of a procedure printed out on the screen
- Usually a procedure is called as part of a more complex set of instructions
- Therefore, usually want to “pass” the results of a procedure call on to other operations, or to store them in variables for later use

e.g.

```
define simple(num1,num2)->result;
   num1+num2->result;
enddefine;


define simple2(num1,num2)->r1->r2;
   num1+num2->r1;
   num1-num2->r2
enddefine;
```

- In the above, **result**, **r1**, and **r2** are known as **output locals** or **result variables**

- When Pop-11 exits(returns) from a procedure the values of the output locals are left on the stack **in reverse order**

e.g.
```
simple(5,6)->x;
x=>
** 11
simple(5,6)=>
** 11

simple2(8,7)->x->y;
x=>
** 15
y=>
** 1
simple2(8,7)=>
** 1 15
```
Note order of results

- **N.B.** => prints top item on stack if used **inside** a procedure, but prints complete stack contents (from bottom up) if used **outside** a procedure

```
define simple2(num1,num2)->r1->r2;
   num1+num2->r1;
   num1-num2->r2
enddefine;
```

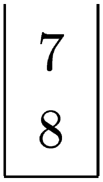
is equivalent to

```
define simple2();
lvars num1, num2, r1, r2;    ;;; all lvars variables
   ->num2;                   ;;;get values for input
   ->num1;                   ;;;locals off stack
   num1+num2->r1;            ;;;same body
   num1-num2->r2
   r2;                       ;;;put values of output
   r1;                       ;;;locals on stack
enddefine;
```

- **Note** that this makes **explicit** the fact that num1, num2, r1, and r2 are all **local lvars**

Why?

- A procedure call like **simple2(8,7)** means
  1) First(i.e. before calling the procedure) evaluate the arguments from left to right (leaving values on the stack)

     7

     8

     Note order reversal here, second argument is on top
  2) Call simple2. It clearly can only get the second argument first, then the first!! So simple2's internals make sure this happens correctly.

- To understand the order reversal with the output values we have to consider what a typical call to simple2 might look like:

  simple2(8,7)–>x–>y;

- This is made up of
  - the call itself
  - two assignments –>**x** and –>**y** (done after the call)
- Note that the assignment to x is done first, followed by the assignment to y
- Therefore the value for x must be on top of the stack, with the value for y below it
- i.e. inside simple2, r2 must be pushed on the stack first, and r1 second – the order is reversed from that in the definition, but a typical call will "match" the header so we don't have to worry about it

More Modern Notation

```
define simple2(num1,num2)->r1->r2;
   num1+num2->r1;
   num1-num2->r2
enddefine;
```

can (should?) be written as follows:

```
define simple2(num1,num2)->(r1,r2);
   num1+num2->r1;
   num1-num2->r2
enddefine;
```

It can then be called as follows:

```
simple2(5,6)->(a,b);
```

and a will get the value of r1, and b the value of r2.

Shortcuts

- This mechanism of leaving results on the stack means we can take “shortcuts”

e.g.
```
define simple(n1,n2)–>res;
   n1+n2–>res
enddefine
```

which leaves the value of res (= n1+n2) on the stack, behaves identically to

```
define simple(n1,n2); ;;;no output local
   n1+n2
enddefine;
```

which will just leave the value of n1+n2 on the stack directly

- Only use the second technique for simple procedures where it is obvious what is being left on the stack, and where. For more complicated procedures be **explicit** i.e. use the first method
- **Do not mix methods in a procedure**

The Return Statement

- Results can also be returned from procedures using the **return** statement

e.g.
```
define simple(num1,num2);
   return(num1+num2)
enddefine;

define simple2(num1,num2);
lvars r1, r2;
   num1+num2->r1;
   num1-num2->r2;
   return(r2,r1)
enddefine;
```

- A **return** statement causes an exit from the procedure immediately (after any values in the brackets have been computed and left on the stack)
- **Again** Do not mix return methods!

Declaring Variables Using vars

- Variables declared by a **vars** statement, whether inside a procedure or not, always have a “box” created for them, unless a “box”with this name already exists
- **Once created this “box” will then exist for the whole of the rest of the Pop-11 session**
- Such a variable(named box) can then be accessed **from anywhere**
- If the **vars** statement occurs in a procedure, then this “box” is created at the time the procedure is **defined**
- All this occurs whether the **vars** statement is inside a procedure or not

Local vars Variables

- If a variable is declared (using **vars**) inside a procedure it is called a **(dynamic) local variable** of the procedure
- **vars** local variables have their **current** value **saved** (somewhere!) on entry to the procedure i.e. *every time the procedure executes*
- **vars** local variables have their previous(saved) value **restored** on exit from the procedure i.e. *every time the procedure returns to where it was called from*
- Therefore, you can safely assign values to **vars** local variables inside a procedure without affecting their value outside the procedure
- They behave (almost!) like new temporary variables

Guidelines

- Use **vars** to declare **global** variables i.e. variables that represent truly global information in your program, that you want anything to be able to access.
- Use **lvars** to declare variables inside a procedure to make them **really** local, rather than the sort of "pretend" locality that **vars** gives.
- **BY DEFAULT:** Input and output locals are **all lvars**
- The general rule is that local variables should always be **lvars** , unless the variable is going to be used as a "matcher variable" (see later in course), or unless there is some other **very** good reason to make it **vars**

IS MSc
AI Programming II

Topic 4

Conditionals

Control Flow

- Normal execution of the statements in a program is **sequential** i.e. the statements are executed in the order they appear in the program.
- Three main ways of altering the flow of control
  - Procedure calls (already met)
  - Conditional statements
  - Loops
- Conditional statements are used whenever we want to carry out some actions depending on whether or not some condition is true or not

Conditional Statements in Pop-11

- There are 3 main forms of conditional statement

(1)
```
if <condition> then
    <actions>
endif;
```

(2)
```
if <condition> then
    <actions>
else
    <actions>
endif
```

(3) Multiconditionals
```
if <condition> then
    <actions>
elseif <condition> then
    <actions>
elseif <condition> then
    <actions>
⋮
else <actions> ;;; this is optional
endif
```

- Also variants using **unless** …**endunless**

Examples

- **if** <*condition*> **then** <*actions*> **endif**

e.g.

```
define positive_difference(num1,num2);
   if num1>num2 then
      num1-num2
   endif
enddefine;

positive_difference(4,2)=>
** 2
```

- **if** <*condition*> **then** <*actions*>
  else <*actions*> **endif**

e.g.

```
define abs_val(num)->result;
    if num<0 then
        -num->result
    else
        num->result
    endif
enddefine;

abs_val(3)=>
** 3
abs_val(-5)=>
** 5
```

e.g.

```
define largest(num1,num2)->res;
    if num1>num2 then
        num1->res
    else num2->res
    endif
enddefine;

largest(7,9)=>
** 9
```

Two Useful Procedures

- **readline** converts what is typed in by the user to a **list**. It prompts the user with a "?" and then anything the user types before hitting <RETURN> is made into a list.

e.g.
```
vars answer;
readline()->answer

? hello there <RETURN>

answer=>
** [hello there]
```

- **member** can be used to test whether or not some item is in a list or not. It returns <true> or <false>.

e.g.
```
member(3,[1 5 6 cat 3 a])=>
** <true>
member("dog",[1 5 6 cat 3 a])=>
** <false>
```

Multiconditionals

-
  ```
  if <condition> then
     <actions>
  elseif <condition> then
     <actions>
  elseif <condition> then
     <actions>
  ⋮
  else <actions> ;;; this is optional
  endif
  ```

e.g.

```
define reply(ans)->rep;
   if ans=[nobody loves me] then
      [why do you feel unloved]->rep;
   elseif ans=[I hate computers] then
      [do you like people]->rep
   else [please tell me more]->rep
   endif
enddefine;

readline()->sentence;

? I hate computers <RETURN>

reply(sentence)=>
** [do you like people]
```

Conditionals and the Stack

- Like everything else in Pop-11 conditionals involve the stack

e.g.

```
if <condition> then
    <actions>
else
    <actions>
endif
```

The <*condition*> can be any Pop-11 expression which leaves a result on the stack. The **if** statement **pops this result off the stack**, and if it is **not** <**false**> then the <*actions*> after the **then** are executed, otherwise the **else** <*actions*> are executed.

- **Note** Any Pop-11 value (data object) apart from <false> is treated as if it were <true> by conditional statements

More Complex Conditions

- Complex conditions can be built out of simpler ones using **and**, **or**, and **not**.

e.g.
```
if x<3 and y>5 then ...
if not(x=y) then ...
if x=3 or (x>75 and y<z) then ...
```

- Conditional statements of the form

```
if not(<condition> ) then
   <actions>
endif
```

can be replaced by

```
unless <condition> then
   <actions>
endunless
```

- Similarly

```
unless <condition> then
   <actions>
else
   <actions>
endunless
```

- In multiconditionals **elseif** and **elseunless** can be mixed
- If a multiconditional starts with **if** it ends with **endif**
- If a multiconditional starts with **unless** it ends with **endunless**

The Pop-11 Pattern Matcher

- A built-in (infix) procedure for matching lists
- Returns a boolean result (<true> or <false>)

e.g. [1 2 3] matches [a b c]=>
** <false>

e.g. [1 2 3] matches [1 2 3]=>
** <true>

- For examples like these (i.e. where the lists involve no pattern variables) it is **much better** to use =
- The patterm matcher should be used when you want to compare a list against a pettern, and to bind variables to values
- Two kinds of pattern variable ? and ??

? Pattern Variables

- ? variables can match a **single** item in a list
- After matching the variable following the ? gets the matching object as its value

e.g. [the cat sat on the mat] matches

```
                    [the ?x sat on the ?y]=>
** <true>
x=>
** cat
y=>
** mat
```

- **Pattern variables must be declared as vars type variables**

?? Pattern Variables

- ?? variables are used to match against **zero or more** items. The relevant variable gets as its value a **list** of the matching items

e.g.

```
[the pretty tabby cat sat on the mat
                         drinking milk]
matches
[the ??description sat on the ?object
                  drinking ??thing]=>
** <true>
description=>
** [pretty tabby cat]
object=>
** mat
thing=>
** [milk]
```

- **Pattern variables must be declared as vars type variables**

- ?? variables always start by matching zero items, then if the whole pattern fails to match the matcher will try again, but this time with the ?? variable matching 1 item, then if this fails, 2 items, and so on

```
e.g. [a b c] matches [??x ??y]=>
     ** <true>
     x=>
     ** [ ]
     y=>
     [a b c]
```

- We can use pattern variables more than once in a pattern

```
e.g. [a b a] matches [?x b ?x]=>
     ** <true>
     x=>
     ** a
```

- Patterns are just lists and so can be **computed**

```
e.g. [the mat]->list;
     [the cat sat on the mat] matches
                              [??x sat on ^^list]=>
     ** <true>
     x=>
     ** [the cat]
```

- If we want to match against a pattern but **don't care** what values actually occur in part(s) of the list we can use = instead of a ? variable, and == instead of a ?? variable

```
e.g. [the castle on the hill] matches
                              [the = on ==]   =>
     ** <true>
```

The Forced Match Arrow -->

- **matches** is typically used as follows:

  if list matches [the ??y on ?z] **then** . . .

- In this kind of use the match may either succeed or fail, and the boolean result of the match is tested by the **if** statement

- In some cases we know **for sure** that a match will succeed, and hence do not need to test the result. In this case we can use the **forced match arrow** -->

- In this case we are really using the matcher to do a complicated form of assignment!

e.g.
```
[a b c d e]-->[?x b c d ?y]; ;;;no result
x=>
** a
y=>
** e
```

- If a forced match fails we will get a **mishap**

Restriction Procedures

- We can restrict the kinds of values allowed to match pattern variables by using **restriction procedures**
- The simplest case is when the restriction procedure returns a Boolean result - See TEACH MATCHES for full information
- **Example**

```
define iscol(item);
   member(item,[red blue green black])
enddefine;
```

```
[the red cat] matches [the ?x:iscol cat]=>
** <true>
[the fat cat] matches [the ?x:iscol cat]=>
** <false>
```

- For a match to succeed the restriction procedures must return true when passed the value of the corresponding pattern variable as a parameter

IS MSc
AI Programming II

Topic 5

Loops

Loops

- Used when we want to repeat some actions over and over again
- The most basic form is the **repeat** loop

```
repeat
   <actions>
endrepeat;
```

- This will execute the <*actions*> over and over again until we interrupt it (using <CTRL>c) or one of the <*actions*> causes an exit from the loop.

e.g.

```
repeat
   3=>
endrepeat;
** 3
** 3
** 3
.
.
.
```

- Another form of the **repeat** loop specifies how many times the loop <*actions*> are to be repeated

```
repeat <expr> times
   <actions>
endrepeat;
```

- <*expr*> is an expressionwhich should evaluate to an integer

e.g.
```
define double(list);
   repeat 2 times
      list=>
   endrepeat
enddefine;

double([three blind mice]);
** [three blind mice]
** [three blind mice]
```

While Loops

- Sometimes we require an action to be carried out repetitively(iteratively) **while** (i.e. for as long as) some condition holds.

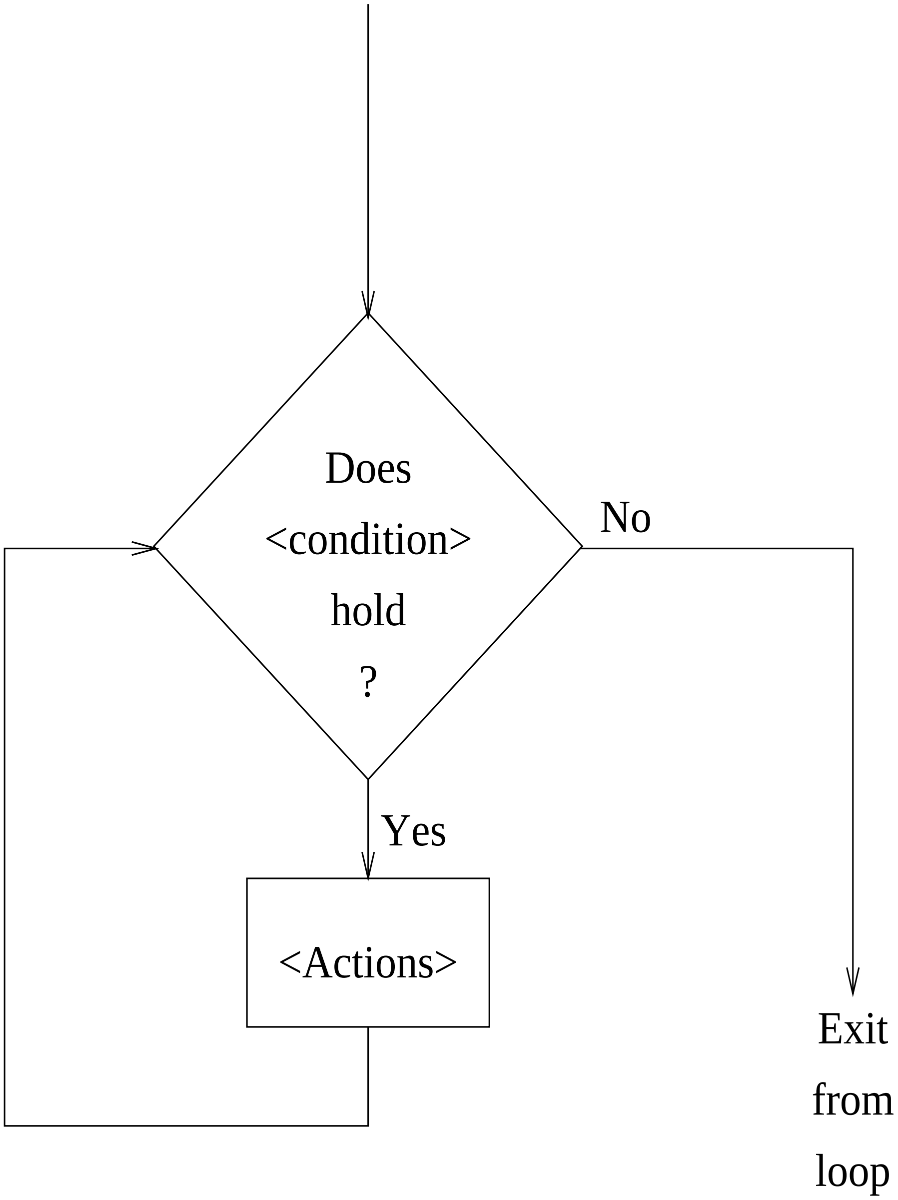
```
while <condition> do
  <actions>
endwhile;
```

e.g.

```
define doubles(num1,total);
  while num1<total do
    2*num1->num1;
    num1=>
  endwhile
enddefine;
```

```
doubles(1,18);
** 2
** 4
** 8
** 16
** 32
```

Flowchart For While Loop



Until Loops

```
until <condition> do
  <actions>
enduntil ;
```

- Equivalent to

```
while not(<condition> ) do
  <actions>
endwhile;
```

e.g.

```
define addone(num1,num2);
  until num1=num2 do
    num1=>
    num1+1->num1;
  enduntil
enddefine;

addone(1,3);

** 1
** 2
```

For Loops

- There are many forms of “for” loop in Pop-11
- One of most useful takes the form:

```
for <element> in <list> do
   <actions>
endfor ;
```

- Used for doing something with/to each element of a list (*iterating down a list*)

e.g.

```
define addnum(num,list);
lvars x;
   for x in list do
      x+num=>
   endfor ;
enddefine;

addnum(2,[1 2 3]);
** 3
** 4
** 5
```

- Another form operating on lists is

```
for <element> on <list> do
   <actions>
endfor ;
```

Sets <*element*> to successive tails of the <*list*> .

e.g.

```
define snip_list(list);
lvars x;
   for x on list do
      x=>
   endfor ;
enddefine;
```

```
snip_list([a b c]);
** [a b c]
** [b c]
** [c]
```

- Another useful form is

```
for <element> from <expr> to <expr> do
  <actions>
endfor ;
```

- The <*expr*>s **must** evaluate to numbers

e.g.

```
for x from 1 to 10 do
    x=>
endfor ;

** 1
** 2
** 3
⋮
** 10
```

- A variant of this is

```
for <element> from <expr> by <expr> to
<expr> do
   <actions>
endfor ;
```

- e.g.

```
for i from 1 by 2 to 10 do
   i=>
endfor ;
** 1
** 3
** 5
** 7
** 9
```

Premature Exits from Loops

One often wants to exit from a loop before it would otherwise end. There are two main ways of doing this.

- Inside a procedure, **return** will exit from the procedure(and hence any loop the **return** is in)
- If one simply wants to exit from the loop **quitloop** can be used

```
e.g. define find(item,list)
     lvars x;
        for x in list do
           if x=item then
              "found"=>
              quitloop
           endif
        endfor
     enddefine;
```

- See TEACH QUITLOOP for information on **quitloop**(n). Also look up **quitif**.

Foreach

- This will look for every instance of a pattern in a list

```
foreach <pattern> in <list> do
    <actions>
endforeach ;
```

- Example

```
[   [mary hates music]
   [john hates jelly]
   [mary loves monkeys]
   [sam hates singing]   ]->info_list;

foreach [?person hates ?thing] in info_list do
   [^person ^thing]=>
endforeach ;
** [mary music]
** [john jelly]
** [sam singing]
```

Forevery

- This “looks” for every instance of a combination of patterns

  forevery <*list of patterns*> **in** <*list*>
  do
    <*actions*>
  endforevery ;

- See TEACH FOREVERY for more details, and limitations

An Example

```
[   [mary hates music]
   [john hates jelly]
   [mary loves monkeys]
   [sam hates music]
   [peter loves prunes]
   [susan hates music]   ]->info_list;

define co_haters(thing, list);
vars p1, p2; ;;;note vars - used in matcher
   forevery [[?p1 hates ^thing]
              [?p2 hates ^thing]] in list do
      [^p1 AND ^p2]=>
   endforevery
enddefine;

cohaters("music", info_list);

** [mary AND mary]
** [mary AND sam]
** [mary AND susan]
** [sam AND mary]
** [sam AND sam]
** [sam AND susan]
etc.
```

Decorated List Brackets

- In a Pop-11 list arbitrary Pop-11 statements may be included between “%” signs
- These statements are “executed” when the code containing the list is run. Anything left on the stack by these statements is then included in the list

```
e.g. [% for i from 1 to 5 do
          i
        endfor %]=>
        ** [1 2 3 4 5]
```

```
[a b c % for i from 1 to 3 do i endfor % d]=>
** [a b c 1 2 3 d]
```

- This is one of the reasons why the user stack is so useful. It enables one to build lists (and other data types) extremely easily and flexibly

IS MSc
AI Programming II

Topic 6

Recursion

Recursion

- Defining objects or actions in terms of themselves
- To build a wall
  1. Build a single layer of bricks
  2. Build a wall on top of it
- Recursive Data Structures

e.g. A **list** is

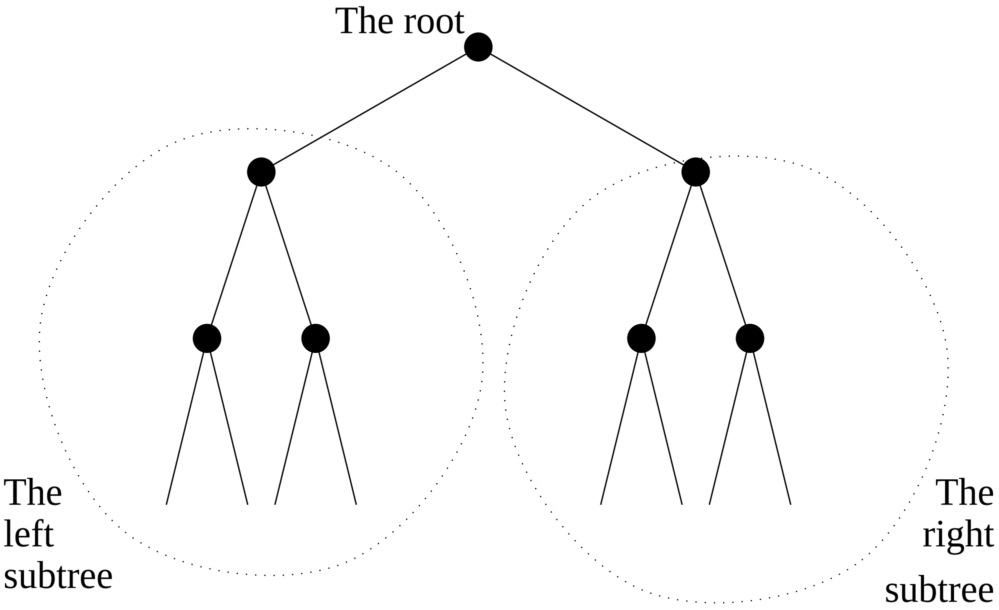
either the empty list
or consists of two parts
- the head(any object)
- the tail *which is a* **list**

- A **binary tree** is

  either the empty tree

  or consists of 3 parts

  - the root(any object)
  - a left subtree a **binary tree**
  - a right subtree a **binary tree**



Recursive Procedures

- Suppose we want to write a procedure **timesup**, which given an integer n as input, computes

  $$1 \times 2 \times 3 \times \ldots \times n$$

  e.g.
  ```
  timesup(3)=>
  ** 6
  timesup(5)=>
  ** 120
  ```

- We could implement this in Pop-11 as:

  ```
  define timesup(n)->answer;
  lvars i;
     1->answer;
     for i from 1 to n do
        i*answer->answer;
     endfor ;
  enddefine;
  ```

- This is an **iterative** solution - it uses a loop

- But notice

  timesup(n)=$1 \times 2 \times 3 \times \ldots \times (n-1) \times n$

  $= \{\ 1 \times 2 \times 3 \times \ldots \times (n-1)\ \}\ \times n$

  = timesup($n-1$)$\times n$

- This gives us a recursive definition of **timesup**

  timesup(1)=1 **stopping case**

  timesup(n)=$n\times$ timesup($n-1$)

- leading to Pop-11 code

```
define timesup(n)->answer;
   if n=1 then
      1->answer
   else
      n*timesup(n-1)->answer
   endif
enddefine;
```

- This is a **recursive** solution

Recursive Definitions

- In general recursive definitions have main parts
  (1) A non-recursive **“stopping”** part
  (2) The recursive part
- Note that either of these may contain several sub-parts
- Note also that (usually!) the recursive part of the definition operates on something “smaller” i.e. nearer to the stopping case

Examples

- The definition of a list splits into two cases
  - The empty list
  - the recursive part (note how the tail is shorter by one element)
- The definition of timesup splits into two parts
  - n=1 (stop by returning 1)
  - n≠1 (return n∗timesup(n-1)) n-1 is nearer 1 than n!!

Local Variables and Recursion

Question How does a Recursive Procedure Keep Track of its Local Variables?

- **Answer**

  Two cases to deal with

  ∗ **lvars** variables

  ∗ **vars** variables

- Local variables **declared by lvars** are created on entry to the procedure. Every time the procedure is called we therefore have new variables which cannot interfere with those in the calling procedure. Therefore when we return back to the calling procedure the variables still have their previous values even though we have in the meantime used **different variables with the same name**

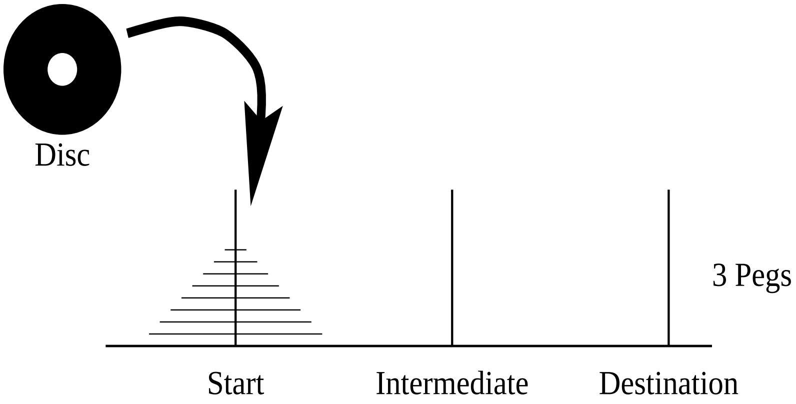
- **Remember** Input and output variables are **lvars** by default.

- Local variables **declared by vars** have their values saved (somewhere) on entry to the procedure. These saved values are restored again on exit from the procedure

  Where are the values saved?
  On another stack known as the **control stack** (**not** the user stack). Because it is a **stack** there is no danger of recursive calls overwriting previous saved values.

- We can override the default (**lvars**) for input and output locals by declaring then as **vars** . It is then possible for them to be accessed (and possibly "interfered with") by procedures called from the one they are declared in. **Only do this for a good reason!!**

The Towers of Hanoi



```
define hanoi(n,start,spare,destination);
   if n=1 then
      [move disc from ^start to ^destination]=>
   else
      hanoi(n-1,start,destination,spare);
      [move disc from ^start to ^destination]=>
      hanoi(n-1,spare,start,destination);
   endif
enddefine;
```

hanoi(3,"peg1","peg2","peg3");

- Recursive solution is short and intuitively obvious(!!)
- Iterative solution needs **deep** insight into the problem

Recursive List Processing

- Because of the recursive nature of lists many procedures which operate on lists are (most) easily written recursively

e.g. Write a procedure **iselement** which returns <true>if a given item is in a list, and <false>otherwise

```
define iselement(item, list) –>result;
   if list=[ ] then
      false –>result
   elseif hd(list)=item then
      true –>result
   else
      iselement(item,tl(list)) –>result
   endif
enddefine;
```

Tracing

- Procedures (especially recursive ones) can often most easily be understood by **tracing** them.

e.g. trace iselement;

(*<ENTER>:trace iselement;* in VED)

```
iselement(3,[4 5 3 1 2])=>

> iselement 3 [4 5 3 1 2]
! > iselement 3 [5 3 1 2]
!! > iselement 3 [3 1 2]
!! < iselement <true>
! < iselement <true>
< iselement <true>
** <true>
```

- Tracing can be turned off again by using

  untrace *procedures*;

  or **untraceall**;

- See TEACH TRACE and HELP TRACE for more details

Reversing A List

```
define reverse(list)->answer;
lvars temp;
   if list=[ ] then
      [ ] ->answer
   else
      reverse(tl(list))->temp;
      [^^temp ^(hd(list))]->answer
   endif
enddefine;
```

```
reverse([a b c])=>
> reverse [a b c]
! > reverse [b c]
! ! > reverse [c]
! ! ! > reverse [ ]
! ! ! < reverse [ ]
! ! < reverse [c]
! < reverse [c b]
< reverse [c b a]
** [c b a]
```

Predicates

- A **predicate** is a procedure taking one argument which returns <true>or <false>

e.g.
```
islist([1 2 3])=>
** <true>

isnumber(3)=>
** <true>

isnumber([1 2 3])=>
** <false>
```

- An object such that a predicate returns <true>when given that object as input is said to **satisfy** the predicate

Another List Processing Example

- Write a procedure **psubset** which takes a list and a predicate as inputs and returns a list of all those elements in the input list which satisfy the predicate

```
define psubset(list,pred) ->result;
lvars temp;
   if list=[ ] then
      [ ] ->result
   elseif pred(hd(list)) then
      psubset(tl(list),pred)->temp;
      [^(hd(list)) ^^temp] ->result;
   else
      psubset(tl(list),pred) ->result
   endif
enddefine;
```

```
trace psubset;

psubset([dog 4 5 cat], isnumber)=>

> psubset [dog 4 5 cat] <procedure isnumber>
! > psubset [4 5 cat] <procedure isnumber>
! ! > psubset [5 cat] <procedure isnumber>
! ! ! > psubset [cat] <procedure isnumber>
! ! ! ! > psubset [ ] <procedure isnumber>
! ! ! ! < psubset [ ]
! ! ! < psubset [ ]
! ! < psubset [5]
! < psubset [4 5]
< psubset [4 5]

** [4 5]
```

psubset — again

```
define psubset(list,pred) ->result;
  if list=[ ] then
    [ ] ->result
  elseif pred(hd(list)) then
    hd(list) :: psubset(tl(list),pred)->result;
  else
    psubset(tl(list),pred) ->result
  endif
enddefine;
```

IS MSc
AI Programming II

Topic 7

Boxes Model of Lists

Some Rather Strange Behaviour

- Consider the following behaviour

```
vars list, list1;
[a b c]->list;
list->list1;
93->list1(2); ;;;change 2nd element
list1=>
** [a 93 c]
```

- This is expected, **BUT**

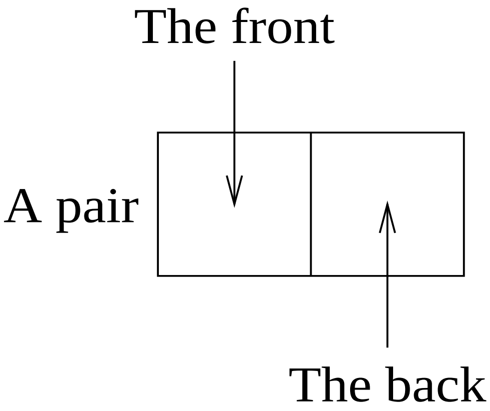
```
list=>
** [a 93 c]
```

- What is going on?

  To understand this kind of behaviour we have to go slightly deeper into the nature of lists than we have so far.

Pairs

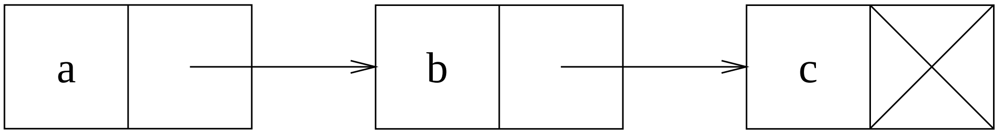
- Lists in Pop-11 (and in Lisp) are built out of more primitive building blocks known as **pairs**.
- Pairs are structures with two components known as the **front** and the **back** of the pair.
- Pairs are represented diagramatically as boxes:



- A 3 element list is actually built out of 3 pairs “chained” together

e.g. [a b c]

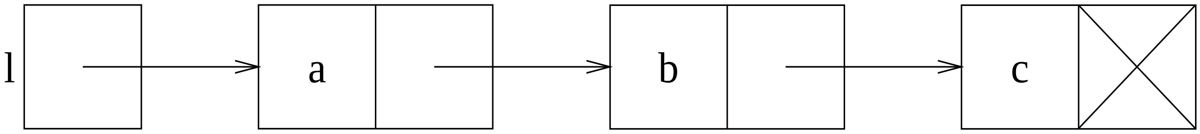
is actually



- **Note** The back of each pair contains a **pointer** to the next
- The last pair contains a special **nil** object as its back
- A list is actually represented as a pointer to a chain of pairs

So [a b c]−>l;

results in



and **the value of l is a pointer**

- **Question** What has happened to the list brackets?
- **Answer** List brackets are purely syntactic constructs present for our convenience.
- The printing procedure does something like:

```
define print(object);
lvars item;
      .
      .
      .
  if islist(object) then
    pr(”[”);
    for item in object do
      print(item);
    endfor ;
    pr(”]”);
  endif;
      .
      .
      .
enddefine;
```

- **Similarly** When the compiler “sees” an opening list bracket ”[” it knows it has to start building a list structure.

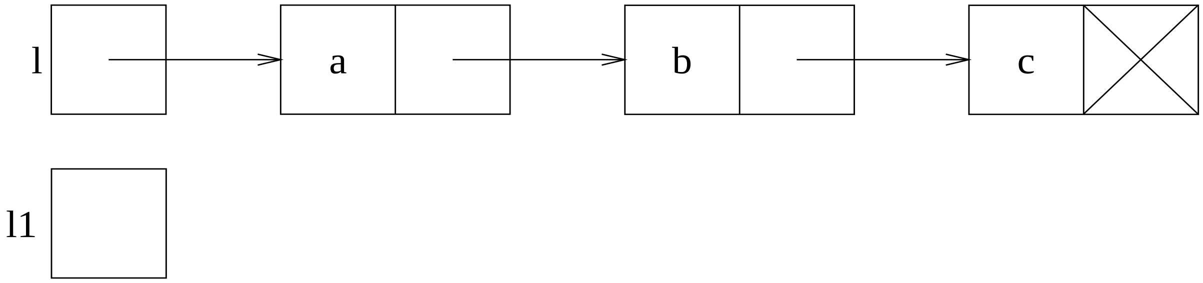
  So for each item following the opening bracket it builds a pair (using **conspair**) with the item in the front, until it reaches the corresponding closing bracket ”]”.

  These pairs chained together form the list

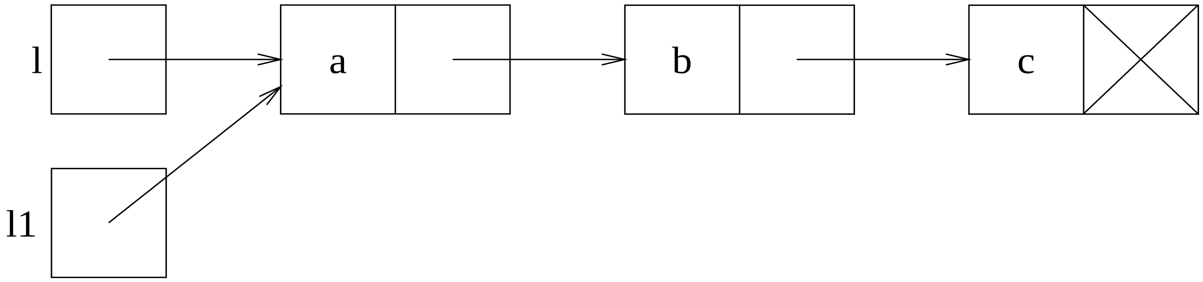
Back to the original example

vars l,l1;
[a b c]−>l;

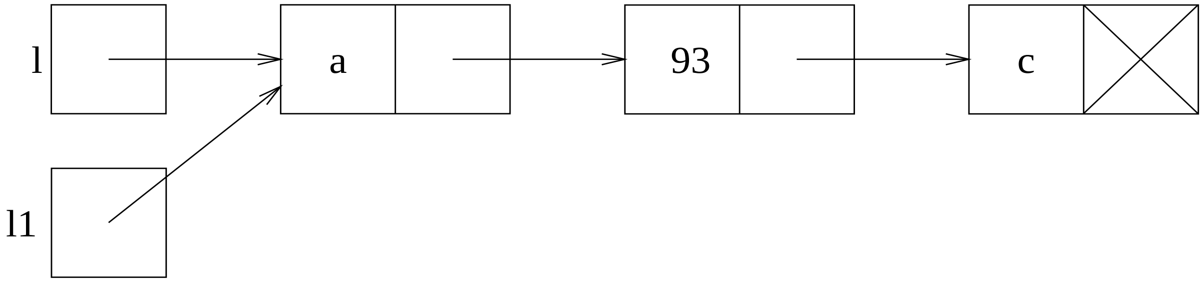
results in



Then doing l−>l1; results in



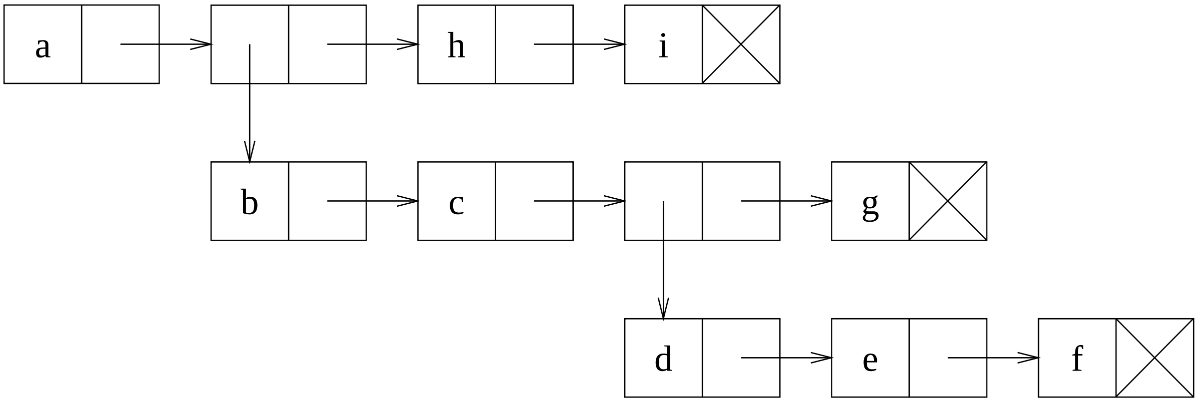
Finally doing 93−>l1(2); results in



- Both l and l1 change because of **structure sharing**

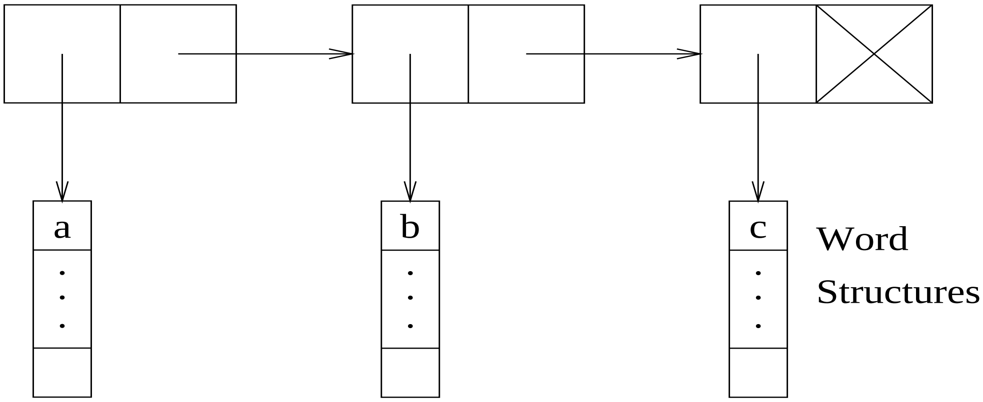
Lists of lists

[a [b c [d e f] g] h i]



A Small White Lie

- [a b c] is really



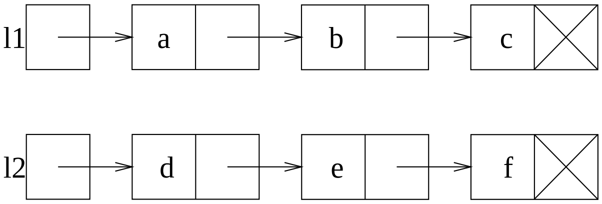
- Where previously we indicated the word ”a” as being “inside” the front of the pair, actually the front of the pair contains a **pointer** to a **word structure**.
- Everything in Pop-11, apart from a few simple things like (smallish) integers is actually represented by **a pointer to an appropriate structure**

To copy or not to copy

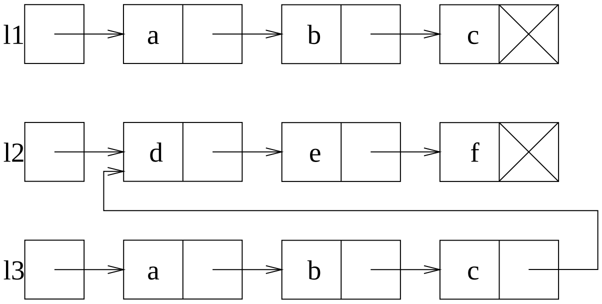
[a b c]−>l1;
[d e f]−>l2;
l1<>l2−>l3; ;;; <> is the concatenation operator

l3=>
∗∗ [a b c d e f]

Before concatenation the situation is:



When doing l1<>l2−>l3; the system **copies l1 but NOT l2**, resulting in:



So changes to l3 can **never** affect l1, but can affect l2!!

- **In general**

  Operations which build new lists out of old ones do **as much copying as is necessary** to ensure that "old" lists do not change, **but no more**

- **Exceptions** are operations which explicitly "side-effect" lists.

e.g. using **hd** or **tl** in updater mode

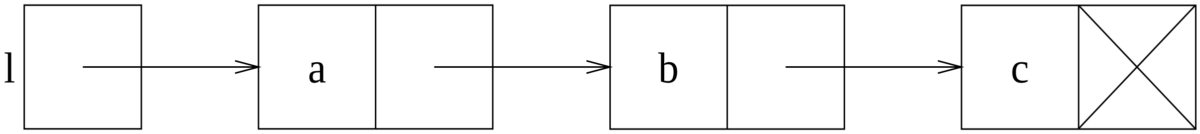
assigning to the second element of a list

etc

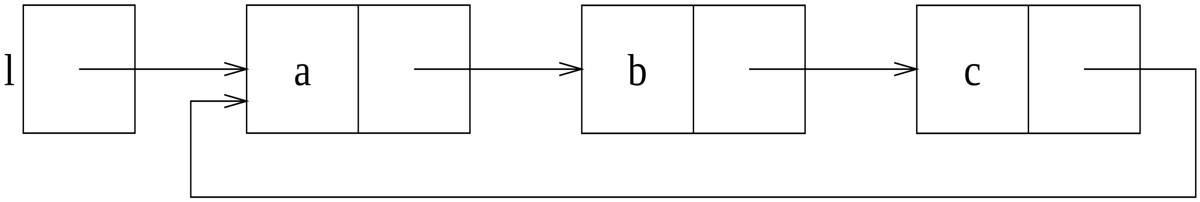
Also versions of operators with **nc_** as a prefix (non-copying)
For example **nc_delete** or **nc_<>**

Circular Structures

[a b c]–>l;



l–>tl(tl(tl(l)));



l=>
∗∗ [a b c a b c a b c a b c............

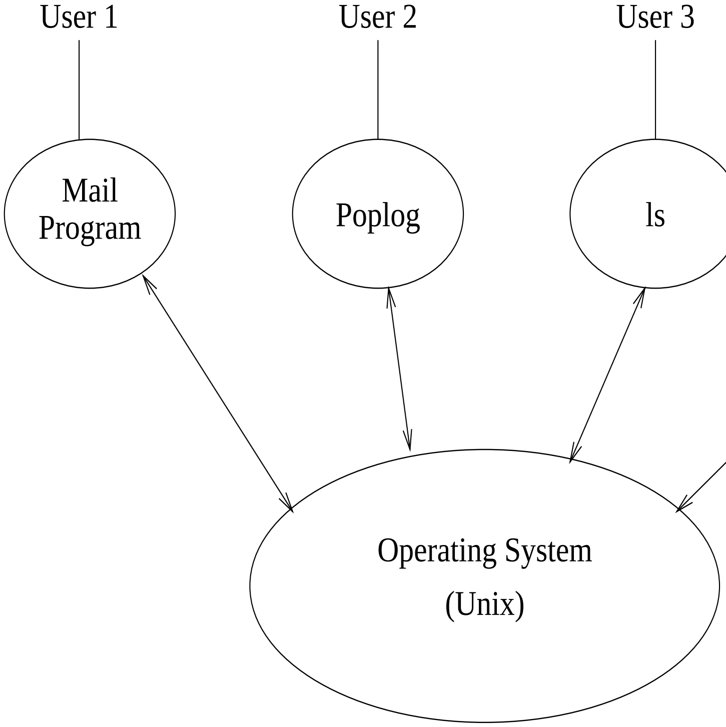
IS MSc
AI Programming II

Topic 8

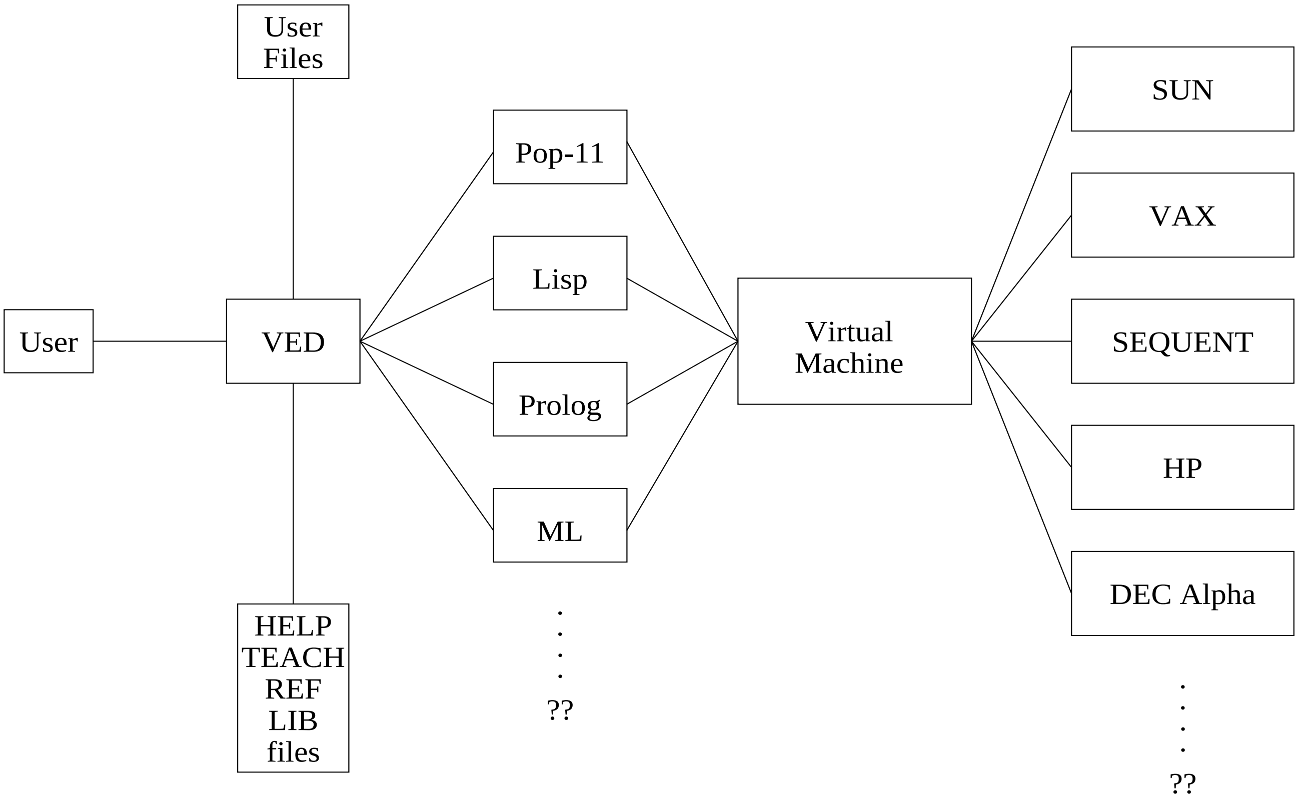
Poplog

Poplog

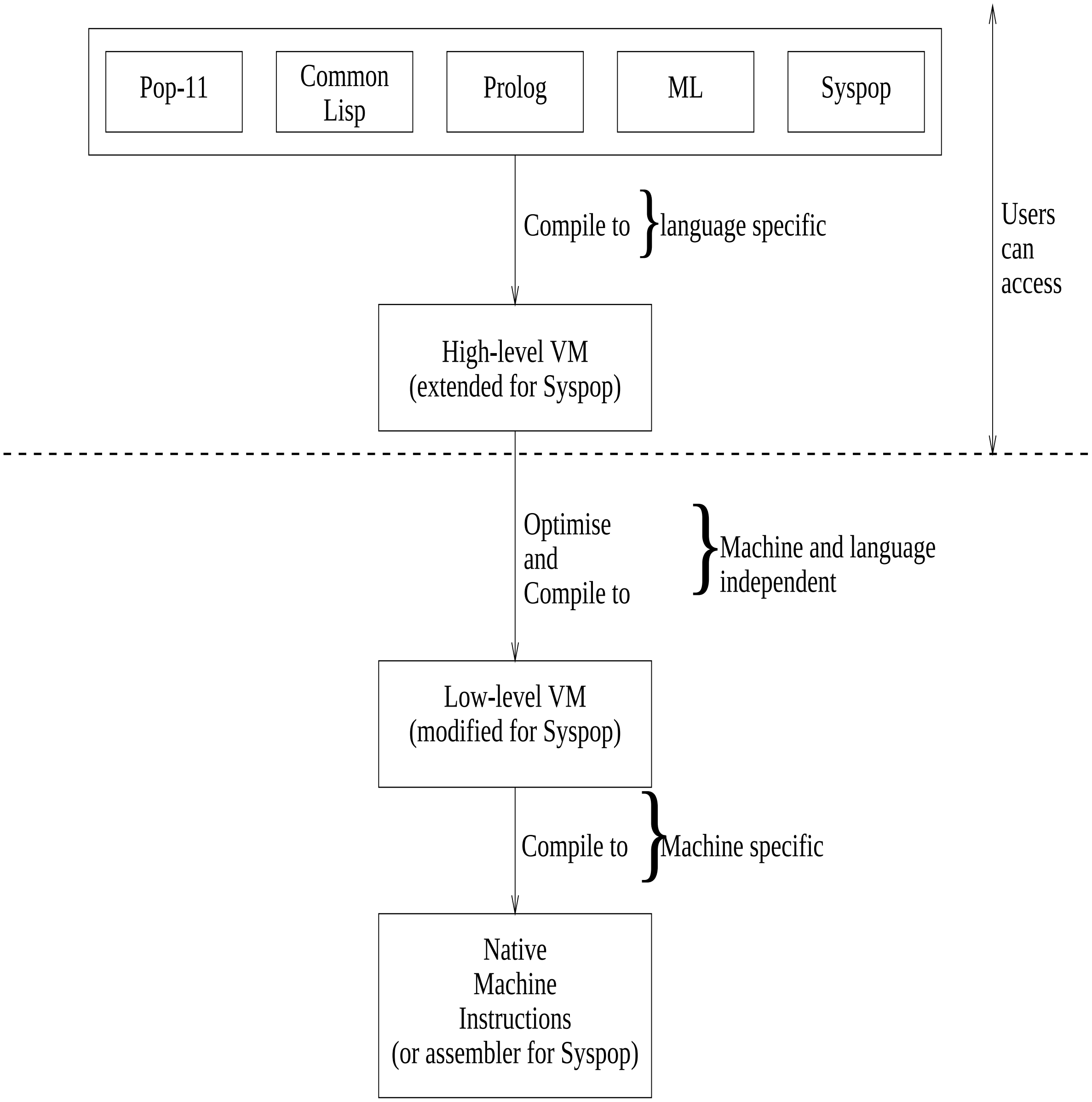
- Poplog is just a **program**
- Runs under control of OS(Unix)
- Asks OS to perform system tasks on its behalf e.g. reading or writing files



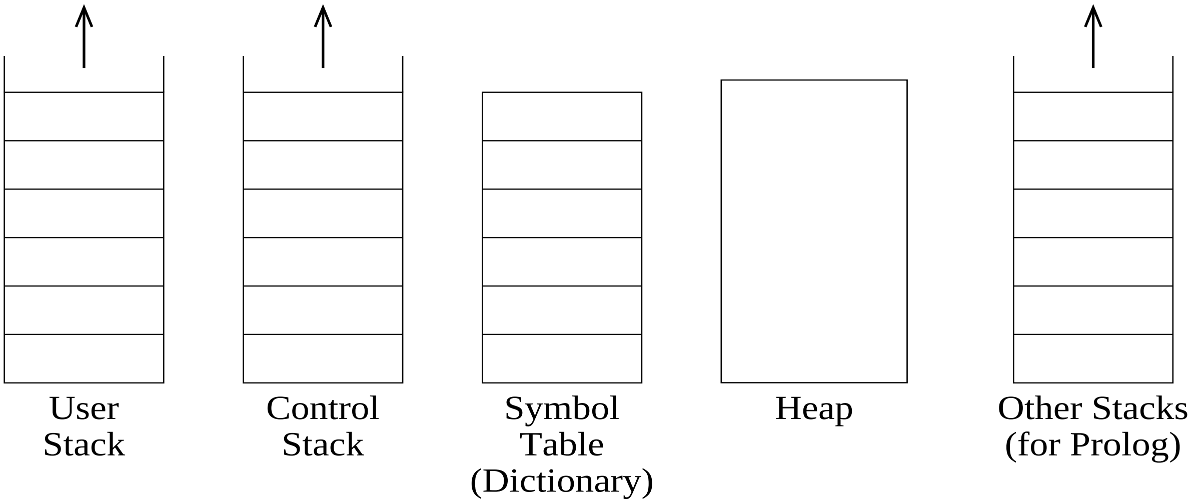
Structure of Poplog


User Files
User
VED
HELP TEACH REF LIB files
Pop-11
Lisp
Prolog
ML
. . . .
??
Virtual Machine
SUN
VAX
SEQUENT
HP
DEC Alpha
. . . .
??

The Compilation Process



The Virtual Machine



- The virtual machine consists of the above + instructions for operating on them

Poplog Virtual Machine Instructions

- Poplog VM instructions are a bit like the following (very) simplified instructions

push	put something on user stack
pop	take something off stack
call	call(execute) a procedure (remembering where it was called)
return	return from a procedure (to where it was called)

- Using these a procedure call is compiled rather as follows:

Pop-11	**VM**		
foo(a,b)−>c;	push	a	;;;1st argument
	push	b	;;;2nd argument
	call	foo	;;;do call
	pop	c	;;; assignment

- A procedure itself is compiled rather as follows:

```
Pop-11                    VM

define foo(a,b)->c;       pop     b     ;;;get 2nd arg
   (a+b)*23->c;           pop     a     ;;;get 1st arg
enddefine;                push    a     ;;;do arithmetic
                          push    b     ;;; ”
                          call    +     ;;; ”
                          push    23    ;;; ”
                          call    *     ;;; ”
                          pop     c     ;;;assign to c
                          push    c     ;;;push result
                          return
```

- Note scope for optimisations

Conditional Statements and Loops

For these we need VM branch instructions

jumpif	*label*	goto *label* if stack top ≠ <false>
jumpifnot	*label*	goto *label* if stack top =<false>
jump	*label*	goto *label*

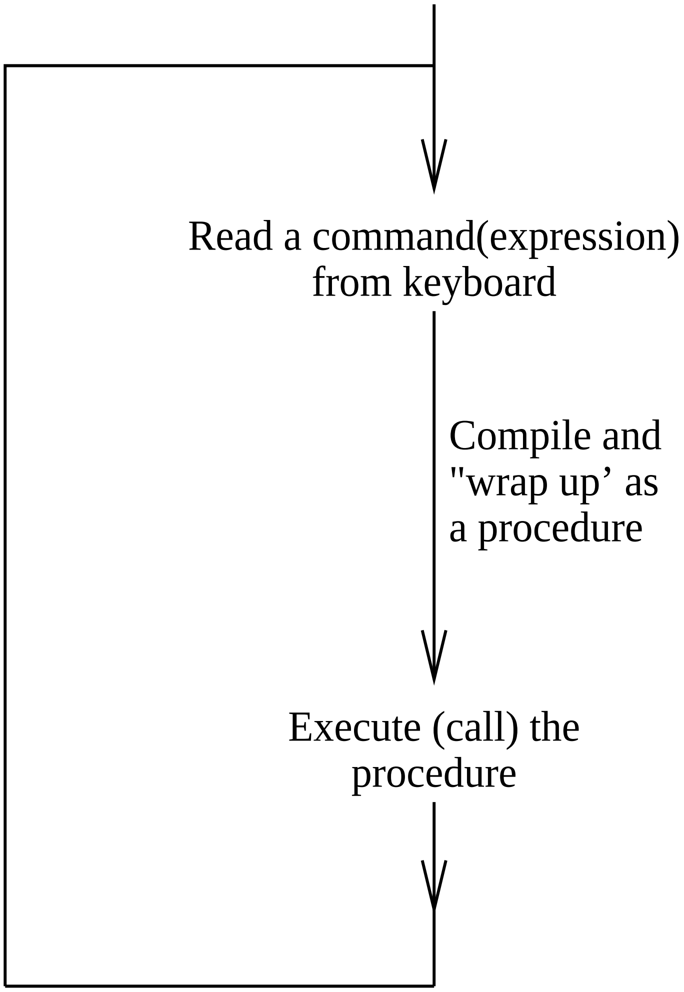
Each of these (except **jump**) also pops the stack.

```
Pop-11                  VM

if x<3 then                   push        x
   5->a;                      push        3
else                          call        <
   2->a                       jumpifnot   l1
endif;                        push        5
                              pop         a
                              jump        l2
                         l1:  push        2
                              pop         a
                         l2:
```

```
Pop-11                   VM

while x<3 do        l1:  push        x
   x=>                   push        3
   x+1->x;               call        <
endwhile;                jumpifnot   l2
                         push        x
                         call        =>
                         push        x
                         push        1
                         call        +
                         pop         x
                         jump        l1
                    l2:
```

- These are not the real VM instructions
- Real ones are similar, but lots more of them, and a bit more complex

Poplog Top-level Loop



Read a command(expression)
from keyboard

Compile and
"wrap up' as
a procedure

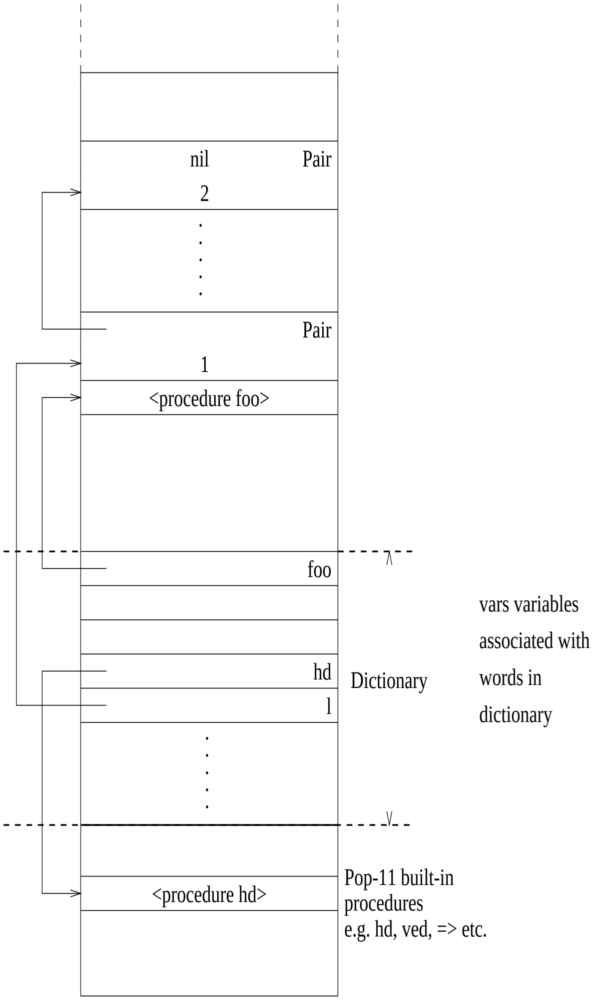
Execute (call) the
procedure

The Heap

The **heap** is where all structures represented by **pointers** (e.g. lists, procedures, vectors etc.) are stored. e.g.

define foo(a,b);
  ⋮
enddefine;

[1 2]–>l;

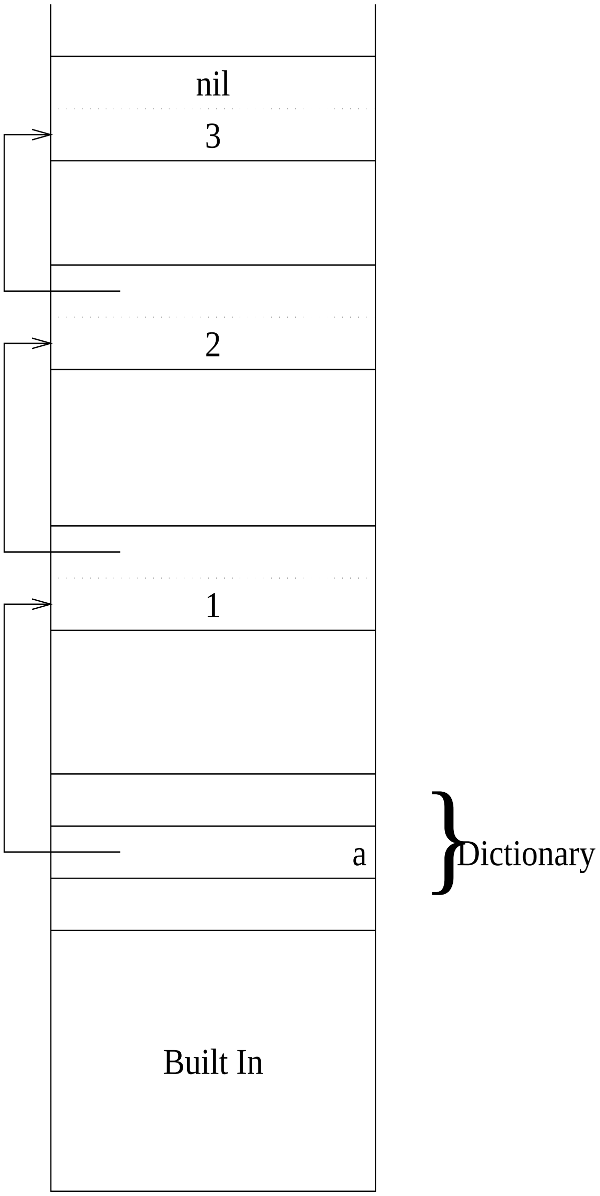


Garbage

vars a;

[1 2 3]−>a;

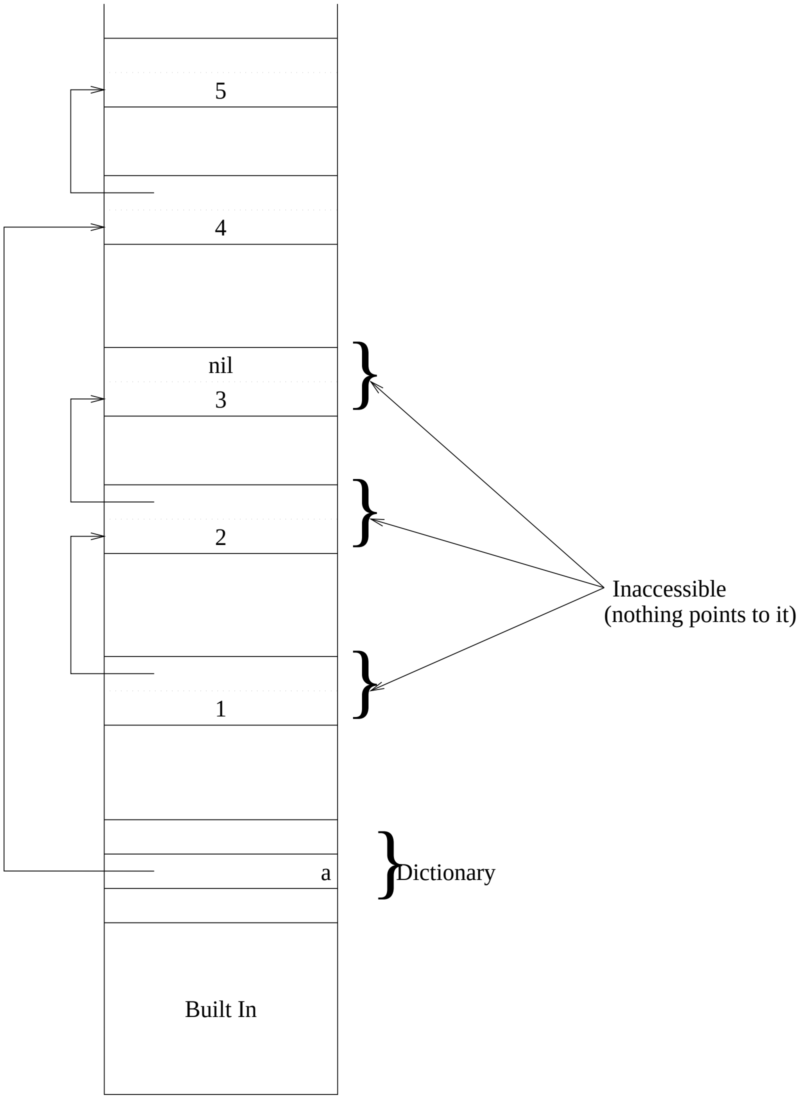
results in



If we then do

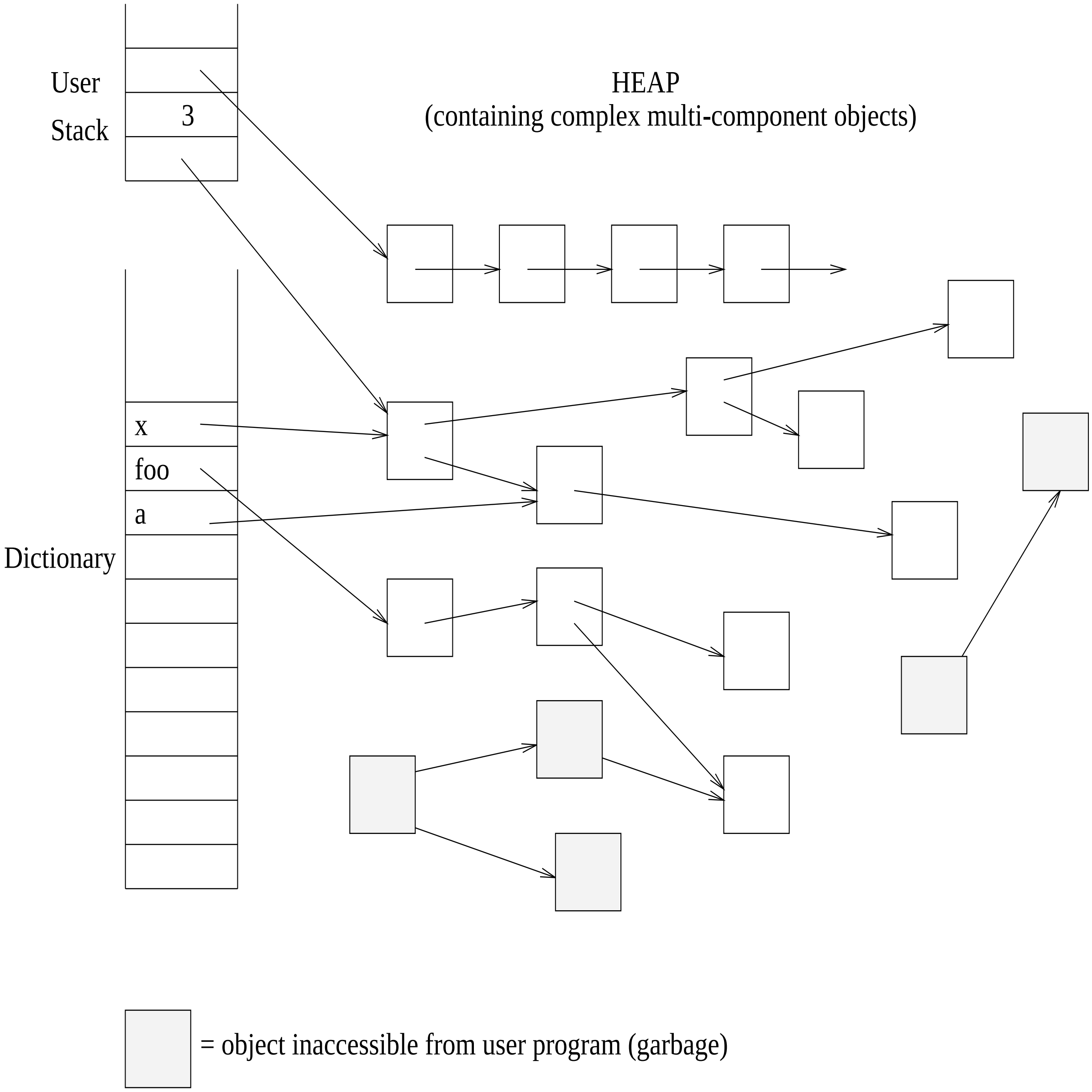
[4 5]−>a;

we get



- Inaccessible objects in the heap are known as **garbage**

The General Situation

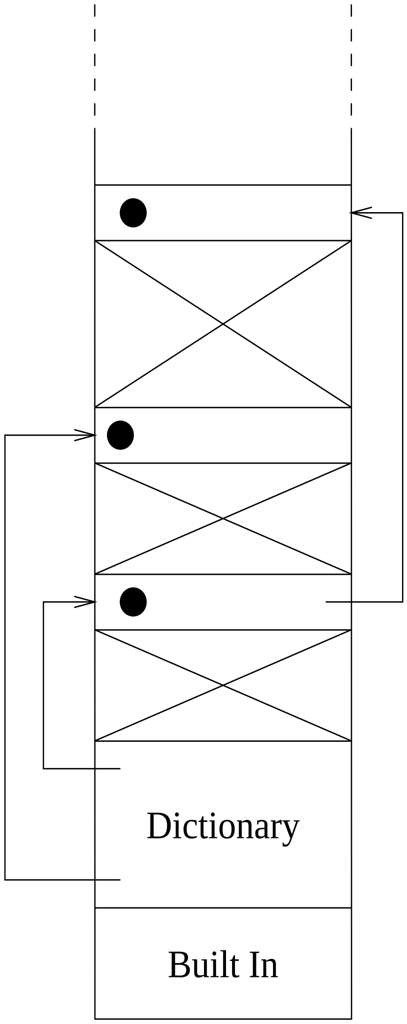


- When POPLOG runs out of space it performs **garbage collection**

Garbage Collection

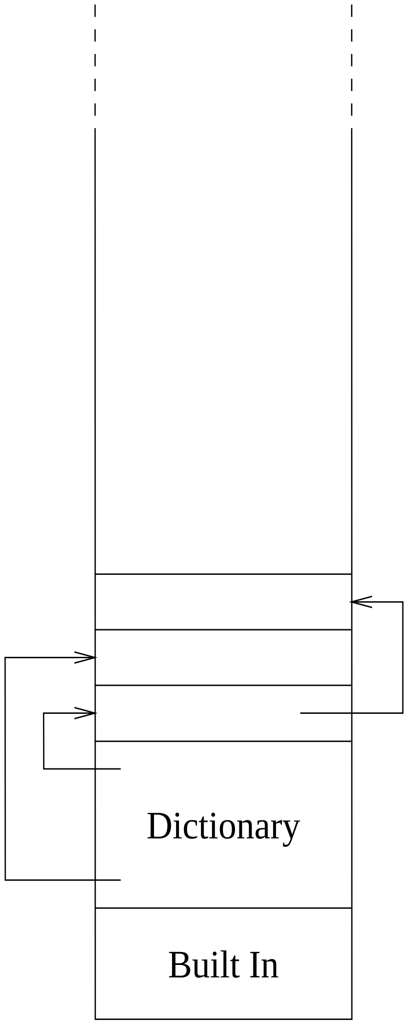
2 of many methods:

- Compacting Garbage Collector



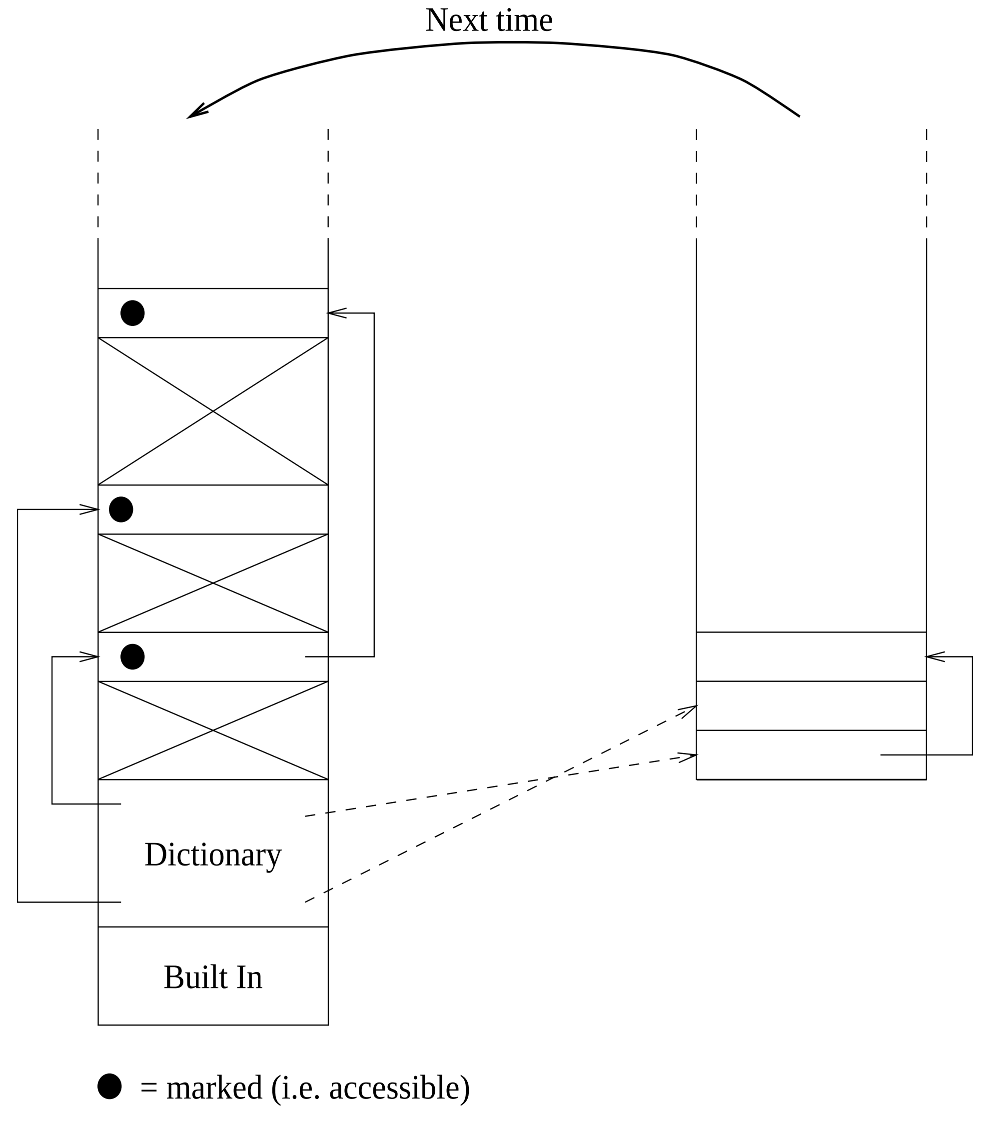
● = marked (i.e. accessible)

Phase 1 - mark all accessible objects



Phase 2 - Compact used memory

- Copying Garbage Collector



IS MSc
AI Programming II

Topic 9

Search

Search

- Many problems in AI can be thought of as search problems

e.g. Theorem proving
Planning
Parsing
Games
Puzzles
⋮

- **This lecture:** How to write searching programs
- See TEACH SEARCHING & TEACH TOWER
- See Thornton & du Boulay (1992): online on COGSWEB

`http://www.cogs.susx.ac.uk/local/help/cogsweb-index.html`

Search Graphs

- In general search spaces are **graphs** rather than trees

e.g. 8-puzzle

- It is therefore important to be able to tell if we have “seen” a state before.
- Therefore our search algorithm will maintain two lists:
  1. **considered** - a list of states that have been looked at already (sometimes called **closed** in the literature).
  2. **alternatives** - a list of states that have been generated but not yet fully examined (sometimes called **open** in the literature).
- The **alternatives** list can be thought of as an **agenda**.

Defining the Problem

- Information about the problem we are trying to solve is given to the search algorithm by means of 4 **problem specific procedures**.

  1. **isgoal(state)** - returns <true> if **state** is a goal state, <false> otherwise.
  2. **isbetter(state1,state2)** - returns <true> is **state1** is "nearer" to a goal state than **state2**, <false> otherwise.
  3. **nextfrom(state)** - returns a list of the "daughter" states of **state**.
  4. **samestate(state1,state2)** - returns <true> is **state1** and **state2** can be considered to be the same as far as the problem is concerned, <false> otherwise.

Choosing a State Representation

- Before writing **isgoal**, **isbetter**, **nextfrom**, and **samestate** you need to decide how you are going to represent a state.
- **Questions**
  - What do you need to know about a state?
  - What needs to be represented **explicitly**?
  - What is OK being represented **implicitly**?
  - What difference would it make if some things that could be held implicitly were represented explicitly?

  - Given a choice of representations, which makes the above procedures:

    easier to write?
    more efficient?
    use less space(memory)?
    create less garbage?
    easier to modify?
    ⋮

- Often a time v. space **trade-off**.
- These are questions you should ask yourself whenever you need to represent anything in a program!!

The Searching Code

```
define search(state);
lvars alternatives, considered, templist;
   [^state]->alternatives;
   [ ] ->considered;
   until alternatives==[ ] do
      dest(alternatives)->alternatives->state;
      state::considered->considered;
      if isgoal(state) then
         return(state)
      endif;
      nextfrom(state)->templist;
      for state in templist do
         unless isoneof(state,alternatives)
            or isoneof(state,considered) then
               insert(state,alternatives)->alternatives;
         endunless
      endfor
   enduntil ;
   return(false);
enddefine;
```

```
define isoneof(state,list);
lvars prevstate;
   for prevstate in list do
      if samestate(state,prevstate) then
         return(true)
      endif
   endfor ;
   return(false)
enddefine;


define insert(newstate,list)->result;
vars state, rest;
   if list matches [?state ??rest]
      and isbetter(state,newstate) then
         insert(newstate,rest)->(result);
         state::result->result;
   else
      newstate::list->result;
   endif
enddefine;
```

Controlling the Search

- To get **depth-first** search

```
define isbetter(oldstate, newstate);
    false
enddefine;
```

i.e. a newstate is always better than an old state, so go on the front of the alternatives list.

- To get **breadth-first** search

```
define isbetter(oldstate, newstate);
    true
enddefine;
```

i.e. an old state is always better than a new state, so new states go on the back of the agenda.

- To get **best-first** search

  Need to define a domain-specific heuristic **isbetter** procedure

  Note: best-first means "best according to the heuristic embodied in **isbetter**" not necessarily the best in absolute terms

- Because of agenda-driven nature of the algorithm it is possible for **isbetter** to change during the search - it is **very flexible**.

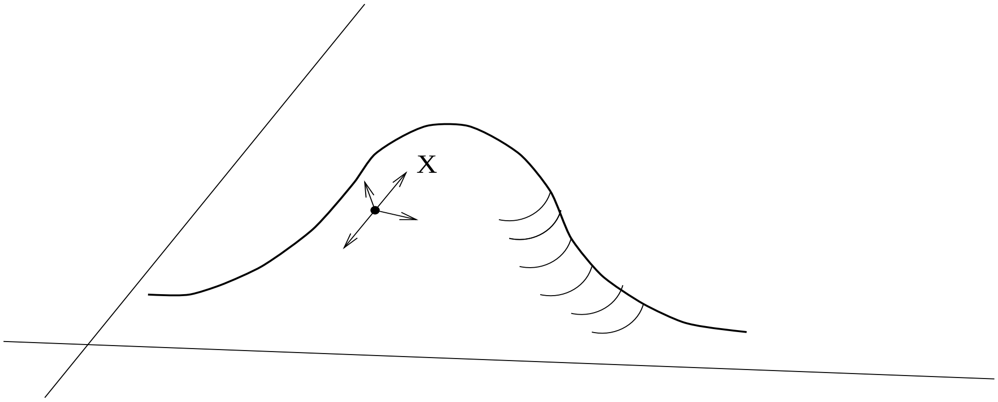
Finding All Solutions

```
define search(state);
lvars alternatives, considered, templist;
   [^state]->alternatives;
   [ ] ->considered;
   [%until alternatives==[ ] do
      dest(alternatives)->alternatives->state;
      state::considered->considered;
      if isgoal(state) then
         state ;;;was return(state)
      endif;
      nextfrom(state)->templist;
      for state in templist do
         unless isoneof(state,alternatives)
            or isoneof(state,considered) then
               insert(state,alternatives)->alternatives;
         endunless
      endfor
   enduntil %]
   ;;;no failure result because [ ] returned
enddefine;
```

- **N.B.** It is possible that this will not terminate

Hill Climbing

Try the "best" place "visible" from where you are



- Problems
  - plateaus
  - foothills
- A variant of depth-first search in which one always chooses the best (according to **isbetter**) of the daughters of the current state to explore next can be regarded as hill climbing

Code for Hill Climbing

```
define search(state);
lvars alternatives, considered, templist;
  [^state]->alternatives;
  [ ] ->considered;
  until alternatives==[ ] do
    dest(alternatives)->alternatives->state;
    state::considered->considered;
    if isgoal(state) then
      return(state)
    endif;
    nextfrom(state)->templist;
    ;;;make list of previously unseen states
    [%for state in templist do
      unless isoneof(state,alternatives)
        or isoneof(state,considered) then
          state;
      endunless
    endfor %]->templist;
    ;;;order these states
    syssort(templist,isbetter)->templist;
    ;;;now put on front of agenda
    templist<>alternatives ->alternatives;
  enduntil ;
  return(false);
enddefine;
```

IS MSc
AI Programming II

Topic 10

Pop-11 Data Structures

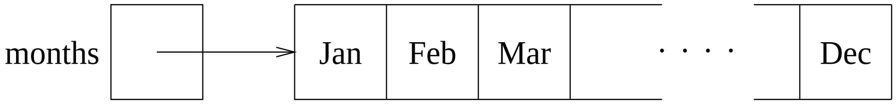
Pop-11 Datatypes

- Have already met:
  - decimals e.g. 5.3
  - integers e.g. 7
  - booleans e.g. <true> <false>
  - pairs (used primarily for building lists)
  - nil (a unique item [])
  - words e.g. ”cat”
  - strings e.g. ’cat’
  - procedures e.g. <procedure hd>
  - lists e.g. [the 3 cats] (a derived type)
  - vectors e.g. {the 3 cats}

- There are lots of others:
  - arrays
  - properties
  - user defined record types
  - user defined vector types
  - keys
  - bignums and ratios and complex numbers
  - closures
  - dynamic lists
  - devices
  - processes
  - refs
  - ⋮
- REF DATA contains a complete list

Vectors

{Jan Feb Mar Apr ...Dec}->months;



- Accessing components of vectors

e.g. months(3)=>
∗∗ Mar

- Updating components of vectors

e.g. ”March”->months(3);

- To make an “empty” vector of length n

e.g. initv(n)->vec; ;;;make vector
5 ->vec(i); ;;; update an element

- Can use vectors whenever we want a fixed length structure. Can use **length** to find out length.

- Can use %...% inside vectors (as well as ^ and ^^).

Strings

- A special type of vector with ASCII codes (numbers representing characters) as elements

e.g.

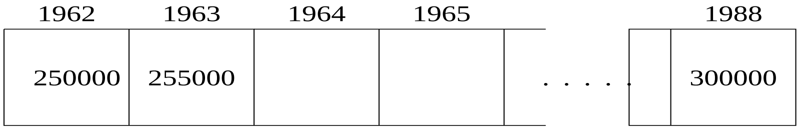
```
'hello rudi'->str;
length(str)=>
** 10           (the space counts)

str(8)=>
** 117        (the code for u)

116->str(8);
str=>
** hello rtdi

inits(n)->str; ;;;creates a string of 0's
```

- See HELP STRINGS

Arrays

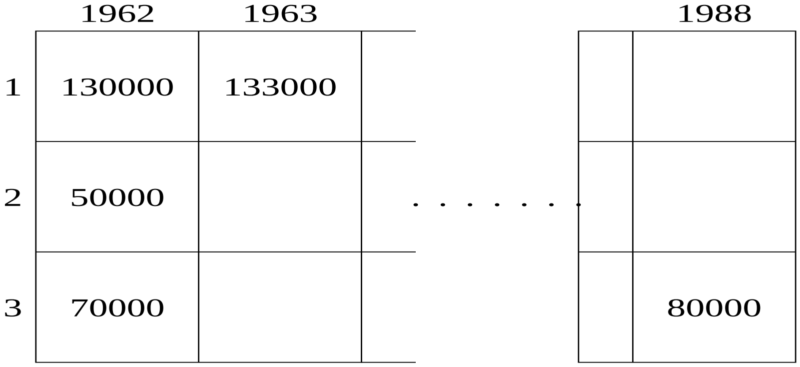
- Arrays are “multi-dimensional” structures

Example 1 A 1-dimensional array



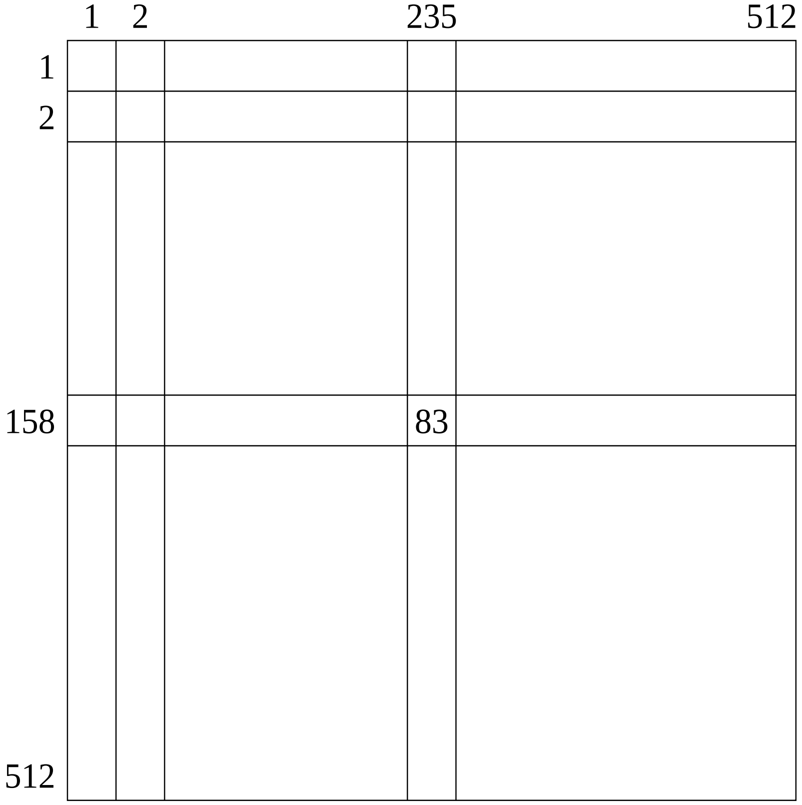
```
newarray([1962 1988])->population;
250000->population(1962);
255000->population(1963);
   ⋮
300000->population(1988);
```

Example2 A 2-dimensional array



```
newarray([1962 1988 1 3])->population;
130000->population(1962,1);
   ⋮
80000->population(1988,3);
```

Representing Pictures



```
newarray([1 512 1 512])->picture;

picture(235,158)=>
** 83
```

- **See**

  HELP NEWARRAY

  (HELP NEWANYARRAY)

Properties (Hash Tables)

- Properties are **efficient** association tables
- Create using:

newassoc	simple	limited
newproperty	less simple	more general
newanyproperty	complicated	very flexible

- **See:**
  - HELP NEWASSOC
  - HELP NEWPROPERTY
  - HELP NEWANYPROPERTY

Newassoc

```
newassoc([[sue 33] [mary 56]])->age;
newassoc([[sue 15000] [mary 17500]])->salary;
age(”mary”)=>
** 56
salary(”sue”)=>
** 15000
18000->salary(”mary”);
age(”sue”)+1->age(”sue”);
age(”joe”)=>
** <false>
3->age(”joe”); ;;;can add new entries
```

Newproperty

```
newproperty([[rudi 44] [ruth 14]], 100,
                    "unknown", true) ->age;

age("rudi")=>
** 44

age("joanna")=>
** unknown
```

- Newproperty has 4 arguments
  1. A list of initialisations (can be [ ])
  2. A “size” for the property. This is **not** a limit on how many items can be stored in the property. More efficient if it is (say) 1.5 times the maximum number of items to be stored in the property.
  3. The default value
  4. To do with garbage collector (**true** is safe value). (If it is **false** then the garbage collector will collect all item/value pairs whose item part is only accessible via the property.)

Appproperty

- **appproperty**(<*property*> ,<*procedure*>)

  applies the <*procedure*> to every item/value pair in the <*property*> . The <*procedure*> should take two arguments, the first for the item, the second for the value.

- Example

```
newproperty([[rudi 44] [ruth 14]], 100,
                          false, true) −>age;

define myprint(x,y);
lvars x, y;
   x=>
   y=>
enddefine;

appproperty(age,myprint);
** ruth
** 14
** rudi
** 44
```

A Warning

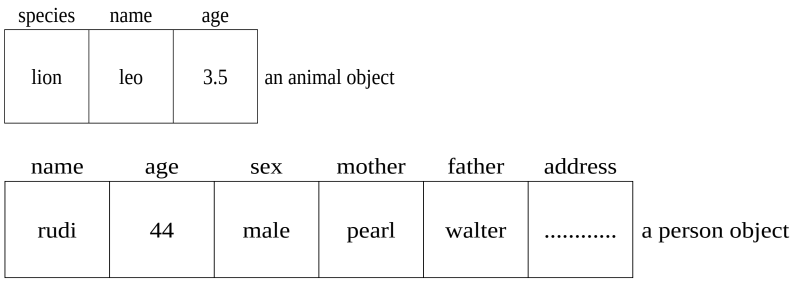
Properties essentially use == when looking items up. Therefore items which are = but not == to items in the property <u>won't</u> be found. To get round this use **newanyproperty**. For this purpose **newmapping** is often simpler.

- See HELP NEWMAPPING
- See HELP NEWANYPROPERTY

Recordclass

- See HELP RECORDCLASS
- We often want to be able to create objects with a set of named fields

e.g.



- **recordclass** is the simplest method

e.g. recordclass animal species name age;

  - tells the system that we are defining a new class of object - the **animal** class.
  - tells the system that objects of this class will have 3 fields.
  - creates various procedures for manipulating objects of this class.
  - creates a **key** for this class of objects (held in variable **animal**_key).

- What procedures does **recordclass** create?

  cons**animal** - for creating animal objects

  dest**animal** - for “taking apart” animal objects (putting the fields on the stack)

  - is**animal** - for recognising animal type objects

e.g.
```
consanimal("lion","leo",3.5)->x;
x=>
** <animal lion leo 3.5>
isanimal(x)=>
** <true>
destanimal(x)=>
** lion leo 3.5
```
(top-level)

Field Accessing and Updating

- In addition, **recordclass** creates procedures (with the same names as the fields of the type of object concerned) for accessing and updating the fields

e.g. (in previous example) we get procedures

  - **species**, **name**, and **age**

- **These all mishap if given non-animal objects as arguments**

- Examples of use:

```
age(x)=>
** 3.5
age(x)+1–>age(x);
age(x)=>
** 4.5
```

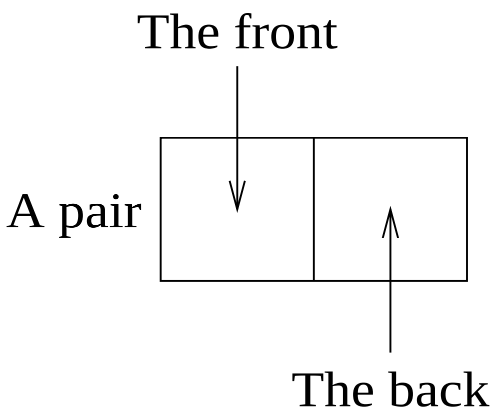
- **Warning:** Different recordclasses should have <u>different</u> field names

- See also HELP VECTORCLASS

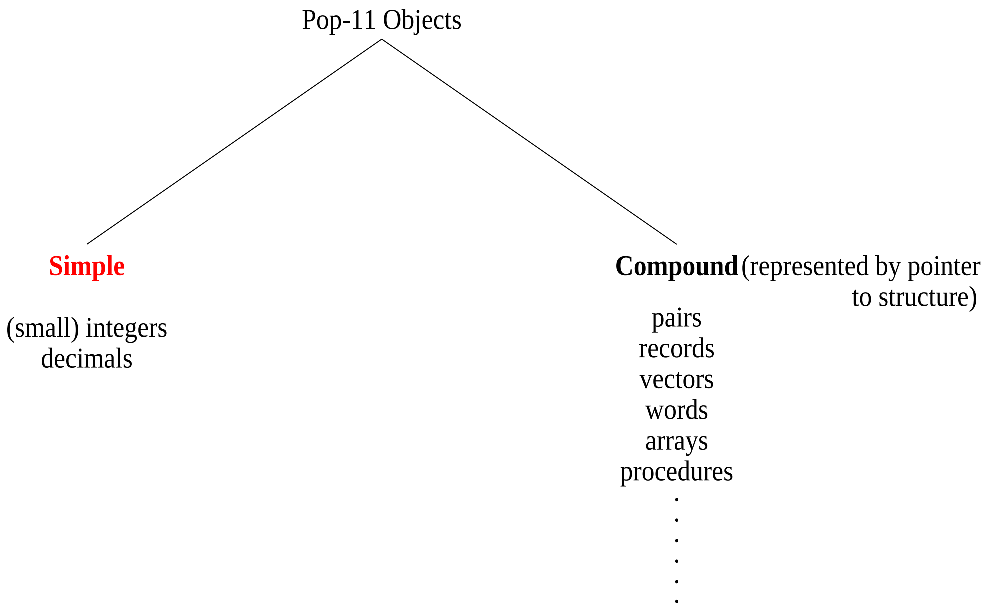
Pairs

- Pairs (used for building lists) are just a kind of record.
- If **pairs** had not existed we could have created them:

  recordclass pair front back;



Pop-11 Objects


Pop-11 Objects

Simple

(small) integers
decimals

Compound (represented by pointer to structure)

pairs
records
vectors
words
arrays
procedures
.
.
.
.
.
.

- Given a simple object and a compound object how does Pop-11 “know” one is simple(not a pointer) and one is compound(a pointer to a structure)?
- All values held in one “machine word”

32	31	30	29	.	.	.	.																					4	3	2	1
0	0	0	0	1	0	0	1																					0	1	1	1

Bits 3 to 32 contain an integer (binary)

If bits 1 and 2 are: 00 then this is treated as machine address i.e. a pointer
01 then this is treated as a single precision floating point value
10 unused
11 then this is treated as an integer

Pointers

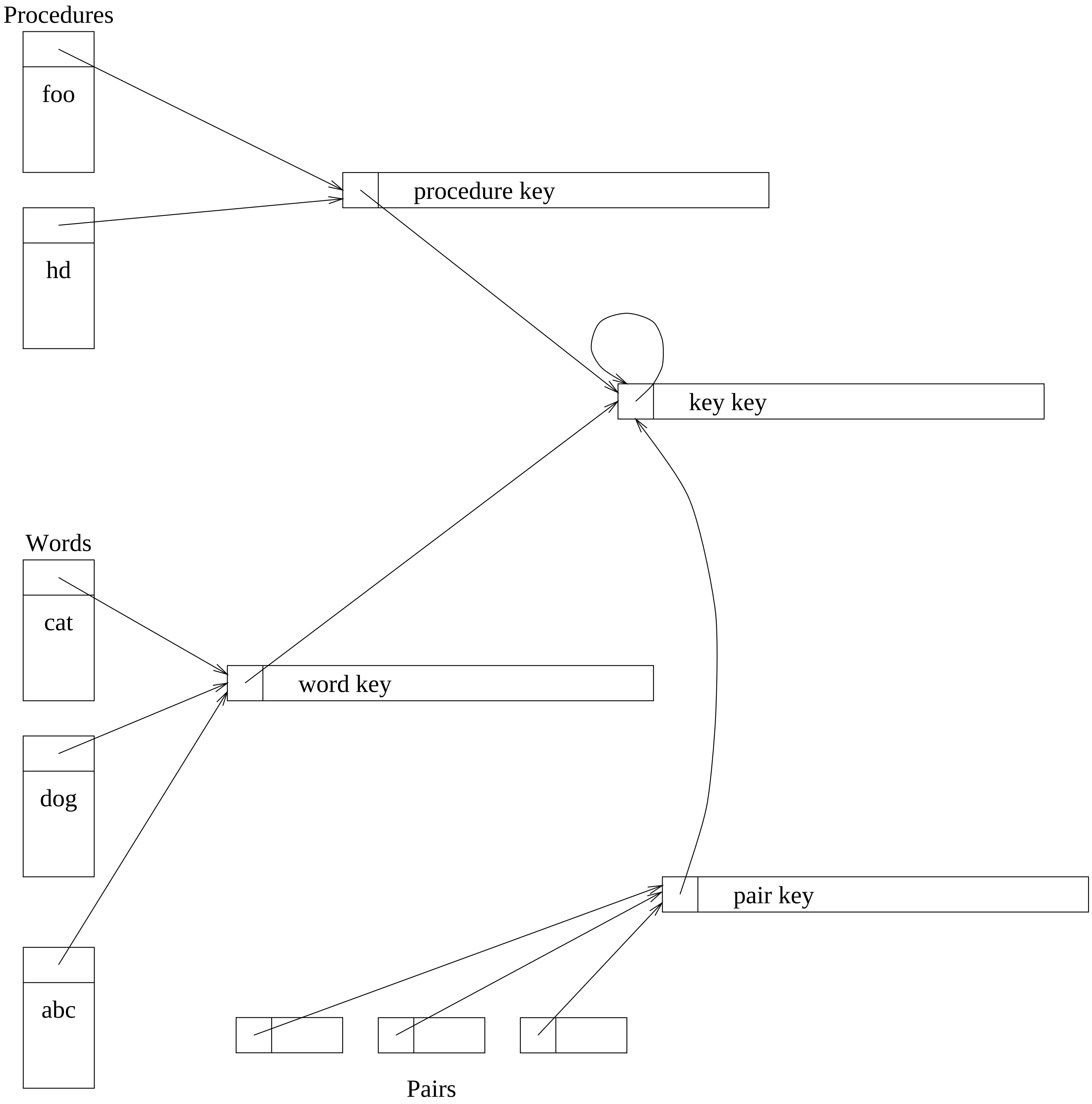
- Given 2 pointers how does Pop-11 “know” that one is a pointer to a procedural object(say) and the other a pointer to a pair(say)?
  - Each object has at its start a special field (called the **key** field which “says” what kind of object it actually is.
  - Actually this field contains a pointer to an object called a **key**.

e.g. All pairs have a pointer to the pair key in this field.

e.g. All words have a pointer to the word key in this field.

etc.

The Key Structure



What's in a Key

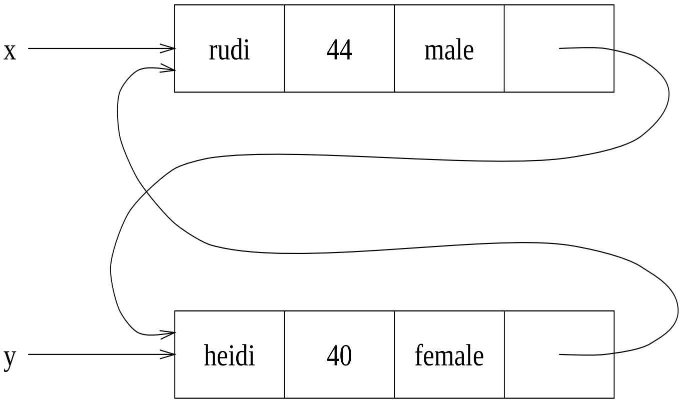
- See HELP CLASSES

class_=	class_access	class_cons	class_apply	class_print	class_dest	class_recognise	.	.	.	....

- Keys contain generic information about the class of objects they are keys for
- For example, the vector key contains:
  - a procedure for testing equality of vectors
  - a procedure for creating new vectors
  - a procedure for "taking vectors apart"
  - a procedure for printing vectors
  - a procedures for recognising vectors
  - a procedure for applying vectors
  - ⋮
- Some of these are user defineable

An example

```
recordclass person name age sex sibling;
consperson("rudi",44,"male",false)->x;
consperson("heidi",40,"female",x)->y;
y->sibling(x);
```



```
x=>
** <person rudi 44 male <person heidi 40
female <person rudi 44 male <person heidi....
```

Makes debugging hard since mishap messages also show this behaviour e.g.

```
MISHAP: List needed
INVOLVING: <person rudi 44 male <person...
```

Never gets to DOING list

Solution:redefine the class_print procedure for person type objects

```
define person_print(x);
lvars x;
   spr("PERSON");
   spr(name(x));
   spr(age(x));
   spr(sex(x));
   spr(if sibling(x)==false then
         false
      else
         name(sibling(x))
      endif);
   npr("ENDPERSON");
enddefine;
```

```
person_print->class_print(person_key);
```

```
x=>
** PERSON rudi 44 male heidi ENDPERSON
```

Recommended Reading

- Chapters 8, 9, and 10 of Laventhol form a good summary of the basic information about data structures in Pop-11
- See HELP CLASSES for more information

IS MSc
AI Programming II

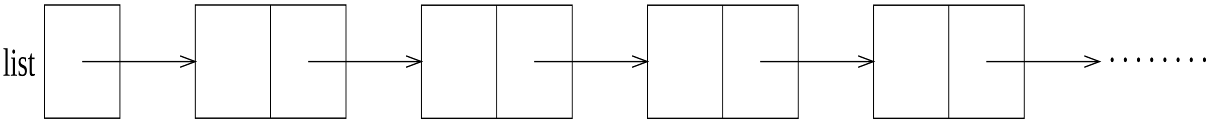
Topic 11

Common Errors

Using Integers to Access Lists in Loops

Consider the following code to add all the elements in a list of numbers:

```
0->sum;
for i from 1 to length(list) do
  sum+list(i)->sum
endfor ;
```



To get at **list**(i) Pop-11 starts from **list** and follows pointers

To get at 1st element it follows 1 pointer

To get at 2nd element it follows 2 pointers

To get at 3rd element it follows 3 pointers

⋮

To get at Nth element it follows N pointer

If the list is N elements long it therefore follows

$1 + 2 + 3 + \ldots + N = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$ pointers.

- What does this mean in practice?
- Assume Pop-11 can do 1,000,000 (10^6) pointer followings per second. Then the times taken to process lists of various lengths are:

Length(N)	$\frac{N^2}{2}$	**Time**
4	8	8μs
10	50	50μs
100	5000	5ms
1000	5000000	$\frac{1}{2}$s
1000000	$\frac{10^{12}}{2}$	139 hours $\approx$ 6 days

- For large lists this takes an unreasonable amount of time!!

- To access each element of a list use **hd** to get at each element, and **tl** to shorten the list each time. **Or**, use the **for** <*item*> **in** <*list*> …construction.
- Both of these only do 2 pointer followings per item, so the above table now looks like:

Length(N)	**2N**	**Time**
4	16	16μs
10	20	20μs
100	200	200μs
1000	2000	2ms
1000000	2000000	2s

- This is a dramatic improvement!!

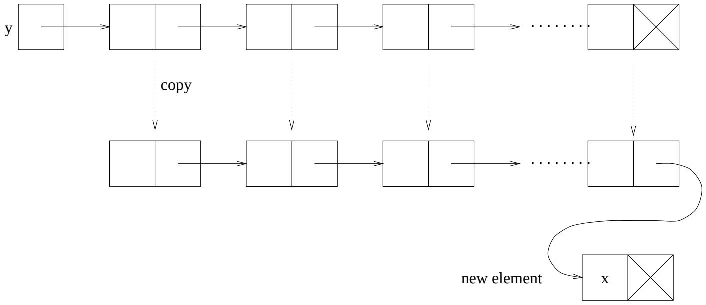
Adding Items to Lists

- Consider the following procedure which builds a list consisting of the cubes of each value in its input list:

```
define cubeall(list)->res;
lvars item;
  [ ] ->res;
  for item in list do
    [^^res ^(item*item*item)]->res
  endfor
enddefine;
```

- This adds items to the end of a list using a [^^y ^x] construction.
- If possible, avoid adding items to the **end** of lists like this (at least repeatedly in loops). Why?

- y is **copied**, and a new pair containing value of x is added at end



- length(y) new pairs are created when copying y. Therefore:

  1st time round loop 0 items copied

  2nd time round loop 1 item copied

  3rd time round loop 2 items copied

  $\vdots$

  Nth time round loop (N-1) items copied

- Therefore, the total number of items copied is $1 + 2 + \ldots + (N-1) \approx \frac{(N-1)^2}{2}$
- **Again** for large lists this can take a very long time

Possible Solutions

- **Solution 1:** Add items at front and reverse afterwards

```
define cubeall(list)->res;
lvars item;
    [ ] ->res;
    for item in list do
        [^(item*item*item) ^^res]->res;
    endfor ;
    rev(res)->res;
enddefine;
```

- **Solution 2:** (even better in this case) Use the stack:

```
define cubeall(list)->res;
lvars item;
    [%for item in list do
        item*item*item
    endfor %]->res
enddefine;
```

Use of <> and ::

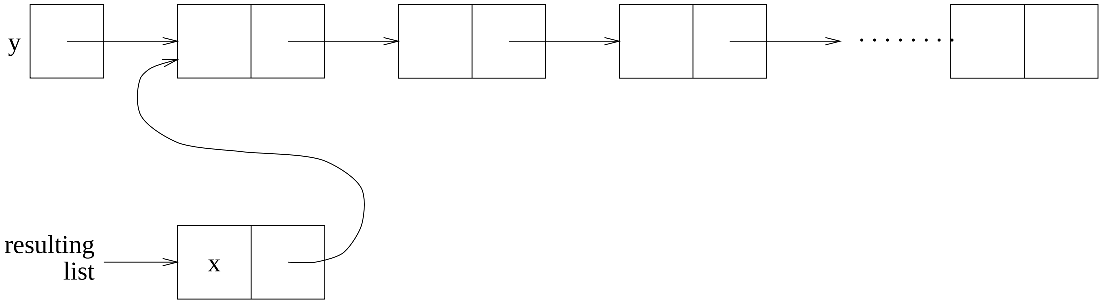
- Do **not** use <> to achieve the effect of ::
- To add a new element (held in **x** to the front of a list (held in **y**) do:

  x::**y**

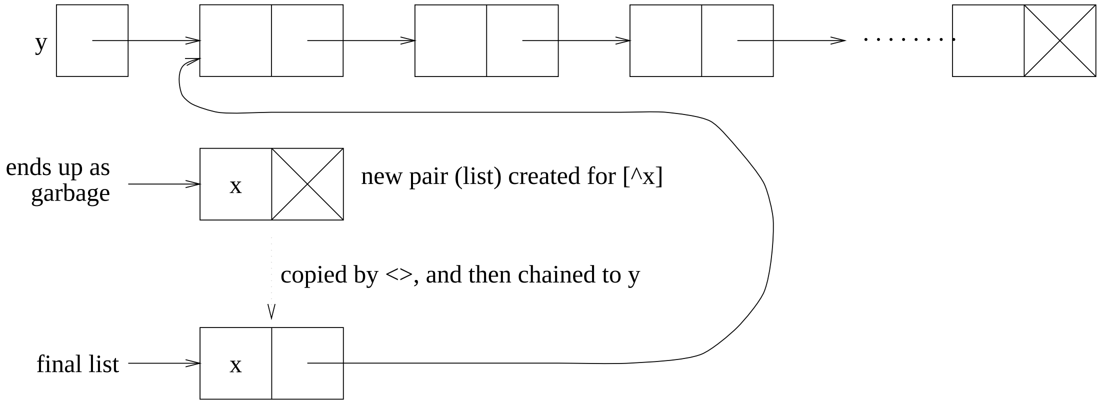
- Do **not** do:

  [ˆ**x**]<>**y**

- **To carry out x::y** *only one* new pair (containing the value of x) is created i.e. the absolute minimum necessary



- **To carry out** [ˆ**x**]<>**y**, an extra **garbage** pair is created since
  1. A list [ˆ**x**] is created.
  2. Then <> *copies its first argument* (the list created in step 1.)
  3. This copy is then “chained” to y, leaving the original pair as garbage (inaccessible).



[^^**x**]

- I often see things like:

  `[^^x]<>result->result;`

- [^^x] is *always* **identically** equal to x
- So you could have written:

  `x<>result->result;`

- This is really just “bad style” indicating some sort of confusion on your part.

True, False, Termin, and Nil

- Pop-11 has several *unique* constants:

  <true>
  <false>
  <termin>
  [ ]

- The above is how they *print*, not how you *refer* to them (except for [ ] of course).

- Pop-11 has *variables*:

  true with *value* <true>
  false with *value* <false>
  termin with *value* <termin>
  nil with *value* [ ]

- These are not the same as the *words* or *strings*:

  ”true” or ’true’
  ”false” or ’false’
  ”termin” or ’termin’
  ”nil” or ’nil’

- **Note:** ’<true>’ is a *string*, **not** the <true> object! **Do not write this and expect to get the <true> object.**

Conditionals and <true> and <false>

- Do not write:

```
if <condition> then
  true
else
  false
endif;
```

- This is (slightly) inefficient, but suggests that you do not understand the stack and booleans properly
- I often see code like:

```
define less_than_three(x)->result;
  if x<3 then
    true->result
  else
    false->result
  endif
enddefine;
```

- This means something like:

  If x<3 evaluates to <true> then put the value of variable **true** (i.e. <true>) on the stack, and then take it off again and assign to **result**. Otherwise, if x<3 evaluates to <false> then put the value of variable **false** (i.e. <false>) on the stack, and then take it off and assign to **result**. On leaving the procedure put the value of **result** on the stack.

- **Note:** this has <true>(say) on the stack, takes it off, puts it on again, takes it off, and puts it on again!!!!

- It is much better to do one of the following:

    1.
       ```
       define less_than_three(x)->result;
           x<3 ->result;
       enddefine;
       ```
    2. (better still in this case)

       ```
       define less_than_three(x);
           x<3
       enddefine;
       ```
- These are all equivalent, but the last is the most efficient, and probably the most natural Pop-11 style

Misuse of the Matcher

- Do not use the matcher when it is easy to use something simpler
- **BAD if** list matches [a b c] **then** ...
  GOOD if list=[a b c] **then** ...
- **BAD if** list matches [=] **then** ...
  GOOD if tl(list)==[ ] **then** ...
- **BAD** list-->[?x ==]
  GOOD hd(list)->x;
- **BAD if** list matches [== ^x ==] **then** ...
  GOOD if member(x,list) **then** ...

Returning Results from Procedures

- **Be consistent (within a procedure) about how you return results.**
- If using an output local make sure that **every** path through the procedure assigns to the variable involved
- If using **return** to return a result, use it on **all** paths through the procedure
- If just leaving things on the stack, then check that you leave something on every path through the procedure(usually!). Comment where you are doing this if the procedure is complicated.
- (Usually) **Do not mix the above methods.** You will avoid errors this way

IS MSc
AI Programming II

Topic 12

Vars and Lvars

An Aside - Anonymous Procedures

- Pop-11 has the facility to define "anonymous" procedures

e.g. **procedure**(x); x+1 **endprocedure**

- This evaluates to a procedural object (left on stack).
- Doing

```
define f(...);
   ⋮
enddefine
```

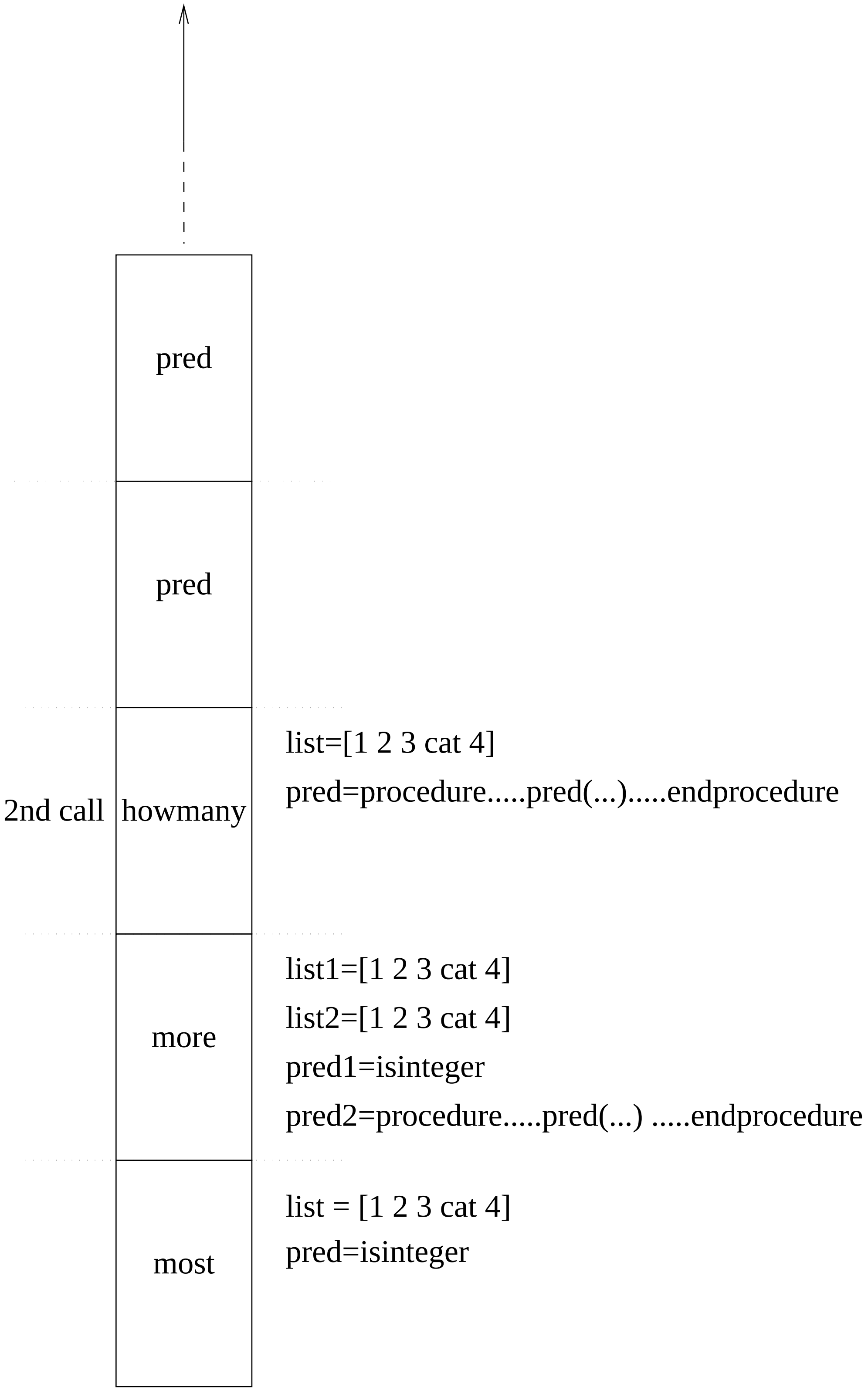
- is (very nearly) equivalent to:

```
vars f;
procedure(...);
   ⋮
endprocedure->f;
```

A Very Nasty Bug

```
define howmany(list,pred)->result;
vars item list pred result;
   0->result;
   for item in list do
      if pred(item) then
         result+1->result
      endif
   endfor
enddefine;

define more(list1,pred1,list2,pred2);
vars list1, pred1, list2, pred2;
   howmany(list1,pred1)>howmany(list2,pred2)
enddefine;

define most(list,pred);
vars list pred;
   more(list,pred,list,procedure(x);
                          not(pred(x))
                       endprocedure)
enddefine;

most([1 2 3 cat 4],isinteger)=>
MISHAP - RECURSION LIMIT EXCEEDED
...
DOING: howmany more most
```

What is going on? There isn't a recursion in sight!!!!


pred
pred
2nd call
howmany
list=[1 2 3 cat 4]
pred=procedure.....pred(...).....endprocedure
more
list1=[1 2 3 cat 4]
list2=[1 2 3 cat 4]
pred1=isinteger
pred2=procedure.....pred(...) .....endprocedure
most
list = [1 2 3 cat 4]
pred=isinteger

Dynamically Scoped Variables (vars)

```
  ⋮
vars x;
2->x;

define foo();
  x=>
enddefine;

define calls_foo();
vars x;
  3->x;
  foo(); ;;;call foo
enddefine;

calls_foo();
  ⋮
```

What is output by this program?

Lexically Scoped Variables

```
    ⋮
lvars x;
2->x;

define foo();
    x=>
enddefine;

define calls_foo();
lvars x;
    3->x;
    foo(); ;;;call foo
enddefine;

calls_foo();
    ⋮
```

What is output by this program?

- See TEACH VARS_AND_LVARS

Full Lexical Scoping

Consider a language with locally defineable procedures, lexically scoped variables, and procedures as “first-class” objects i.e. that can be passed as arguments to, and returned as results from, other procedures

```
define f();
lvars x;
   define g();
         ⋮
         x
         ⋮
   enddefine;
   ⋮
   return(g)
enddefine;

f()->h;
h();
```

- Procedure g has a built-in “reference” to x in f
- If x is held in f’s stack frame this will no longer exist when g is called (via h).
- In other words, the extent of the variable x is “bigger” than its scope.
- Dealing with this (implementing it) so that nothing goes wrong (no “dangling references”) is hard - **Type 3 lexical variables**
- Solution is to recognise when this may happen and have these variables not in the procedure stack frame but in the heap
- See REF VMCODE for more details

Example of Use of Type 3 Lvars

```
define make_counter()->counter;
lvars n=0;
  procedure();
    n+1->n;
    return(n);
  endprocedure->counter
enddefine;
```

```
make_counter()->f;
make_counter()->g;
```

Creates procedures with **private** variables which maintain their value between calls e.g.

```
f()=>
** 1
f()=>
** 2
g()=>
** 1
f()=>
** 3
g()=>
** 2
```

etc.

Guidelines

- If a variable is just a temporary local variable use **lvars** (even for input and output locals), unless you want to use the variable as a pattern variable (after ? or ??) in the matcher.
- For global variables use **vars** while debugging, and change to **lvars** when finished, unless it is used in the matcher.
- If something must be a **vars** variable declare it as **vars** *at top level* and use **dlocal** inside procedures instead of **vars** , to have the variable saved and restored on entry/exit to/from the procedure
- See TEACH VARS_AND_LVARS

Dlocal

Consider the following procedure:

```
define f(...);
vars x;
   ⋮
enddefine;
```

The **vars** declaration does two things:

- At *compile time* creates a global variable x.
- At *run time* causes the value of x to be saved on entry to the procedure, and restored on exit.
- The recommended style is to separate these two functions, and to write:

  ```
  vars x; ;;;global declaration of variable
  define f(...);
  dlocal x; ;;; save and restore
     ⋮
  enddefine;
  ```

Vars Variables and Words

- Every **vars** variable is associated with the Pop-11 *word* of that name.
- Words are represented by pointers to appropriate word records. These contain a **valof** field, used to hold the value of the variable **(not quite the whole story!!)**
- Clearly each **vars** variable needs a unique location. Therefore words are stored in a dictionary and so are unique - when code to create a word is executed the dictionary is consulted, and if the word already exists the existing one is used, otherwise a new word is entered in the dictionary for future use.
- Creating (declaring) a **vars** variable therefore amounts to making sure there is a dictionary entry fo the word involved, and "flagging" it as a variable.

IS MSc
AI Programming II

Topic 13

Advanced Features

Character Repeaters and Consumers

- In Pop-11 characters are **integers** in range 0-255
- A procedure which takes no arguments and produces a character as its result is known as a **character repeater**
- A procedure which takes a a character as its argument and produces no result is known as a **character consumer**
- In Pop-11 all I/O is normally done via character repeaters and consumers

Charin and Charout Etc.

- **charin** - reads a character from the keyboard
- **charout** - sends a character to the screen
- **cucharin** - all Pop-11's input procedures (e.g. **readline**) ultimately use **cucharin**. The default value of **cucharin** is **charin**.
- **cucharout** - All Pop-11's output procedures (e.g. =>, **pr**, etc.) ultimately use **cucharout**. The default value of **cucharout** is **charout**.
- By assigning different values to **cucharin** or **cucharout** we can make Pop-11 take its input from somewhere else (e.g. a file), or send its output somewhere else (e.g. a file).

Discin and Discout

- **discin**(<*filename*>) - produces a character repeater which reads from file <*filename*>
- **discout**(<*filename*>) - produces a character consumer which writes to file <*filename*>
- **Example** - suppose file rudi.foo contains the characters:

  Hello there Rudi aged 44

  discin('rudi.foo')−>f; *;;; build repeater*
  f()=>
  ∗∗ 72 *;;;character code for H*
  f()=>
  ∗∗ 100 *;;;character code for e*
  etc.

  f() produces <termin> at end of file

  Note: Spaces are characters (code 32).
  As a character 3 has code 51

- Similarly:

  discout('temp')−>f; *build consumer*
  f(97); *;;;write character* **a** *to file temp*
  f(98); *;;;write character* **b** *to file temp*
  f(termin); *;;;close file*

- File temp now contains the letters:
  ab

- discout('temp')−>cucharout;
  ensures that all output via =>, pr, etc all goes to file temp

- Make sure that the program closes file temp when it has finished
  e.g. by cucharout(termin);

Item Repeaters and Consumers

- These produce or consume items rather than characters
- **incharitem**(<*character repeater*>)

  produces an item repeater which gets the characters to make up the items using the given <*character repeater*>
- **E.g.** `incharitem(discin(’rudi.foo’))−>f;`

```
f()=>
∗∗ Hello
f()=>
∗∗ there
f()=>
∗∗ Rudi
f()=>
∗∗ aged
f()=>
∗∗ 44
f()=>
∗∗ <termin>
```

  Subsequent calls to f will mishap.

- **Note:** The characters are formed into items using Pop-11's normal itemisation rules. These can be changed - see REF ITEMISE
- **outcharitem** is similar. The item consumers produced will take any Pop-11 object as argument, and send the characters correponding to its printed form to the appropriate place

Writing to Different Files

- Some programs which write different types of information to different files e.g.
  - error messages to an error file
  - real output to an output file
  - progress information to a log file

If the main (top-level) procedure contains code like the following:

```
define main(errorfile,outputfile,logfile);
vars werr, wout, wlog;
   outcharitem(discout(errorfile))->werr;
   outcharitem(dicout(outputfile))->wout;
   outcharitem(discout(logfile))->wlog;
   ...
enddefine;
```

then all procedures can use calls like:

```
   werr([this is an error message]), or
   wlog([The program has reached here])
   etc.
```

to write anything at all to the appropriate file.

Dynamic Lists

- Lists with a procedure for calculating the next element.
- They are constructed from the procedure using **pdtolist**.

e.g.

```
vars n=1;
define f();
   2*n;
   n+1->n;
enddefine;

pdtolist(f)->list;
list=>
** [...]
list(2)=>
** 4
list=>
** [2 4 ...]
hd(list)=>
** 2
tl(list)->list;
list=>
** [4 ...]
```

Proglist

- The compiler always works on a list held in **vars** variable **proglist**
- **proglist** is a dynamic list, containing the items to be compiled.
- At top level, **proglist** is initialised by:

  pdtolist(incharitem(cucharin))–>proglist;

Popval

- **popval** compiles (and executes, if appropriate) the items in a list

  Example

  ```
  vars list, x=3;
  [if x<2 then ”yes”=> else ”no”=> endif]
                                      –>list;
  list=>
  ** [if x < 2 then ” yes ” => else ” no ” =>
  endif]
  popval(list);
  ** no
  ```

- The effect of:

  ```
  load foo.p
  ```

  could have been achieved by:

  ```
  popval(pdtolist(incharitem(discin(’foo.p’))));
  ```

- See HELP POPVAL (and more modern equivalent HELP POP11_COMPILE)

Readitem

- **readitem** returns the first item in **proglist**, and removes it from there
- It is roughly equivalent to:

```
define readitem()->res;
   if null(proglist) then
      termin->res
   else
      hd(proglist)->res;
      tl(proglist)->proglist;
   endif
enddefine;
```

- **Note:** In the above we could not write

  if proglist==[ ] . . .

  since this would not work on dynamic lists. The procedure **null** returns <true>on either an empty ordinary list ([ ]) or on an empty dynamic list (one whose procedure has returned <termin>)

Macros

- Macros are procedures which are executed during compilation. Any results returned are put back on the front of **proglist** and normal compilation is resumed
- The execution is triggered when the compiler encounters the name of the macro in proglist

e.g.
```
define macro swap;
lvars x, y;
   readitem()->x;
   readitem()->y;
   x; ";" ; y ; "->" ; x ; "->" ; y ;
enddefine;
```

- Then writing:

```
swap a b;
```

in a program is exactly the same as writing

```
a;b->a->b;
```

since this is **actually what is compiled**

Another Example

- Macros can be used to define arbitrary new syntax forms

e.g.

```
define macro dotwice;
lvars x;
   ”repeat”; 2; ”times”;
   readitem()–>x;
   until x==”enddotwice” do
      x;
      readitem()–>x;
   enduntil ;
   ”endrepeat”
enddefine;
```

```
dotwice
   [hello there]=>
enddotwice
```

```
** [hello there]
** [hello there]
```

- See HELP MACRO for more information
- Also see Chapter 11 of Laventhol

IS MSc
AI Programming II

Topic 14

More Advanced Features

More about Procedures

- Procedures are complex objects, with several fields that users can get at, e.g.
  - **pdprops**
  - **pdnargs**
  - **updater**
- A variable with a procedure as its value actually has a pointer to the appropriate procedural object as its value (of course!)
- The **pdprops** field is normally used to hold the name of the procedure. For a procedure defined using the **define** . . . **enddefine** syntax, this contains the word (actually a pointer to the word) representing the name of the procedure.
- For a procedure defined using the **procedure**. . . **endprocedure** syntax, the **pdprops** field contains <false>.

Pdnargs

- The **pdnargs** field of a procedure contains the number of arguments that the procedure expects. It is used by such things as the tracing procedure.
- For a procedure defined by the **define** ...**enddefine** syntax the **pdnargs** field is set to the number of arguments that the procedure is defined with.
- See HELP PDNARGS and HELP PDPROPS for more information on these

Updaters

- Consider the following two uses of procedure **hd**:

  (1) hd(x)=>

  (2) 4–>hd(x);

- The first use of **hd** involves a procedure that takes a single argument (a list), and accesses it, putting one result (the first element) on the stack.
- The second use of **hd** involves a procedure that takes two arguments (a list, and a new value for the head), and updates the list (altering the contents of the first pair), and leaves no results on the stack
- Although these two procedures have the same name they are clearly different!
- How does this work?

- A procedure call to the right of an assignment is said to be in *updater mode.*

- A procedure call not to the right of an assignment is just a normal procedure call. A procedure call in updater mode calls a different procedure from a normal call - it calls a *updater* procedure instead

- Updater procedures are stored in the **updater** field of the normal procedure

- Procedures without an updater have <false>in this field

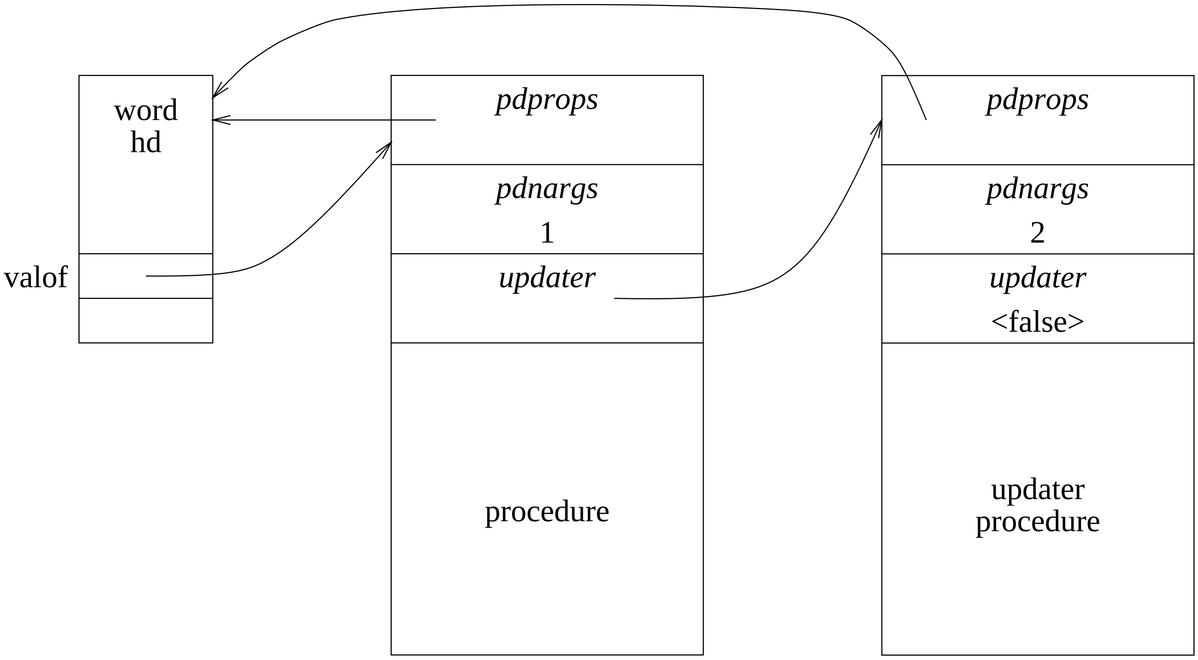
e.g. 7–>sqrt(49)

causes a mishap because the **sqrt** procedure has no updater.

e.g. 4–>hd(x)

is OK because the **hd** procedure has an updating procedure (actually a pointer) in its **updater** field.

- For the **hd** procedure the situation is as shown below:



(Key fields not shown)

Defining Your Own Updaters

- Suppose you have defined a procedure **second** as follows:

```
define second(list)->res;
   hd(tl(list))->res
enddefine;
```

- This enables one to write things like:

```
second(x)->item;
```

- In some situations it might then be nice (to increase code readability etc.) to be able to write things like:

```
100->second(x);
```

- Unless you write an updater for **second** this would mishap (of course!)

- You can write an updater for **second** as follows:

```
define updaterof second(val,list);
   val->hd(tl(list))
enddefine;
```

- You can only do this once **second** itself has been defined.
- Now you can do the following:

```
[a b c]->list;
second(list)=>
** b
5->second(list);
list=>
** [a 5 c]
```

Closures

- Pop-11 supports the notion of “partially applying” a procedure.
- Partial application involves “freezing” some of the arguments to a procedure, resulting in a new procedure requiring fewer arguments.
- Consider the procedure **add3** defined by:

```
define add3(n1,n2,n3)−>res;
  n1+n2+n3−>res;
enddefine;
```

- This procedure takes 3 arguments. If we fix the last to be 5 then we have a new procedure which expects 2 arguments, and which returns the sum of these plus 5

e.g.

```
partapply(add3,[5])−>add2;
add2(1,3)=>
** 9
```

- Similarly we can "freeze" the last two arguments, giving a procedure needing only 1 argument

e.g.
```
partapply(add3,[10 9])->add_nineteen;
add_nineteen(5)=>
** 24
```

- In general **partapply** takes two arguments:
  1. a procedure, taking n arguments.
  2. a list of m values (representing values for the <u>last</u> m arguments of the procedure.
- The result of a call to **partapply** is:
  3. A procedure expecting $n - m$ arguments.

- Any call to **partapply** such as:

e.g. partapply(add3,[10 9])->add_nineteen;

- can also be written with the alternative "nicer" syntax:

  add3(%10,9%)->add_nineteen;

- In Pop-11 partially applied procedures are known as **closures**.
- See **HELP PARTAPPLY**

Defining Infix Operators

- It is sometimes useful to define your own <u>infix</u> operators

E.g. An infix operator to do vector addition

```
define 4 ++(x,y);
lvars i;
   unless isvector(x) and isvector(y)
                  and length(x)=length(y) then
      mishap(’Need Equal Length Vects’,[^x ^y])
   endunless;
   {%for i from 1 to length(x) do
      x(i)+y(i)
   endfor %}
enddefine;
```

```
{1 2 3}->v1;
{8 9 10}->v2;
```

```
v1++v2=>
** {9 11 13}
```

```
1++2=>
;;; MISHAP - Need Equal Length Vects
;;; INVOLVING: 1 2
;;; DOING : ++ ...
```

- Alternative syntax for this is:

  define 4 x ++ y;

  ⋮

  enddefine;

- The number on the **define** line is the precedence of the operation. The higher this number the *lower* the precedence. For instance + has precedence 5 while $*$ has precedence 4, meaning that multiplication is done before addition in the absence of bracketing information.

- See **HELP PRECEDENCE**

- See **HELP OPERATION**

Processes

- A **process** is a Pop-11 data structure that records the state of an execution of a piece of Pop-11 program
- Information stored in a process includes:
  - the state of the call stack
  - the state of the user stack
- There are 2 ways to create processes:
  - consproc
  - consprocto
- **E.g.** consproc(n,proc)−>p;
- n is an integer, and specifies how many items will be taken from the user stack and put on the process' own private user stack at the time it is created
- proc must be a procedure

- A process can be run using **runproc**

  e.g. runproc(n,p);

  where n is an integer specifying how many items are moved from the user stack to the process stack when it is run, and p is a process.

- When a process is run for the first time, the procedure (associated with the process) is called. Subsequently, the procedure starts from where it was last suspended (see below).

- A process can suspend itself using **suspend**

  e.g. suspend(n)

  where n specifies how many items are passed from the process user stack to the main user stack.

- After suspension of a process, control then returns to the program which called **runproc** (or **resume**).
- A subsequent **runproc** on the process will continue from the **suspend**
- If the procedure the process was created with ever exits normally (i.e. is returned from) then <u>all</u> values on the process stack are put on the user stack and the call of **runproc** returns. The process is then marked as "dead". Subsequent attempts to run it will then mishap.
- See **REF PROCESS** for more details.

Summary of Processes

- Create by:
  - consproc
  - consprocto
- Activated by:
  - runproc
  - resume
  - kresume
- Suspended by:
  - suspend
  - ksuspend
- Tests:
  - isprocess
  - isliveprocess

An Example

```
define search(state);
lvars alternatives, considered, templist;
   [^state]->alternatives;
   [ ] ->considered;
   until alternatives==[ ] do
      dest(alternatives)->alternatives->state;
      state::considered->considered;
      if isgoal(state) then
         suspend(state,1) ;;; was return(state)
      endif;
      nextfrom(state)->templist;
      for state in templist do
         unless isoneof(state,alternatives)
            or isoneof(state,considered) then
               insert(state,alternatives)->alternatives;
         endunless
      endfor
   enduntil ;
   return(false);
enddefine;

consproc(init_state,1,search)->get_solution;
```

Then every time we want a new solution, doing

```
   runproc(0,get_solution)->sol;
```

will leave a solution in **sol**.

Non-Standard Control

- Pop-11 has various facilities for altering the normal control flow of a program
  - **exitfrom**
  - **exitto**
  - **chain**
  - **catch**
  - **throw**
  - etc.
- These provide facilities to:
  - exit from the current procedure to a named procedure in the call chain
  - exit from a named procedure in the call chain
  - replace the current procedure call with another one
  - etc.

An Example

```
define f();
    ⋮
    g()
    ⋮
enddefine
define g();
    ⋮
    h()
    ⋮
enddefine
define h();
    ⋮
    k()
    ⋮
enddefine
define k();
    ⋮
    exitfrom(g)
    ⋮
enddefine
```

- Suppose the top-level call is:

  f();

- When **k** is called the call chain consists of:

  k h g f

- The **exitfrom** therefore exits one from the call of **k**, **h**, and **g**, returning to the point in **f** as if **g** had returned normally. All local variables are restored etc properly.

- The same effect could have been achieved by:

  exitto(f);

- See **HELP EXITFROM** and related files

THE END

IS MSc
AI Programming II

Topic 15

Introduction to Lisp

Common Lisp

⋮

Zeta

Franz

MacLisp Scheme

Spice

P-Lisp

McCarthy
Lisp. . . Lisp 1.5
1956. . . 1962

TLC

1984
Common Lisp

UCI

InterLisp Standard

PSL

NIL

XLisp

⋮

Main Differences

- No infix operators in Lisp
- Every expression has parentheses
- (Therefore (**lots (and (lots (and lots)))**
                    (of brackets)
)
)
- Things like the following are valid variable names:

  a-b

  abc

  cat36

  mc^2

- No (user) stack!!

Books

- Steele,G. (1984) *Common Lisp: The Language.* Digital Press

  This is the reference manual for Common Lisp

- Hughes, S. (1986) *Lisp.* Pitman, Computer Handbook Series

  A pocket-sized mini reference book (cheap!)

- Wilensky, R. (1986) *Common Lispcraft.* W.W. Norton and Co.

  A thorough introduction.

- Winston, P.H., and Horn, B.P.K (1984) *LISP*(2nd edition)

  A good introduction to the language and basic AI Programming techniques

On-Line Documentation

- TEACH READLISP
- TEACH CLISP
- TEACH LISPVED
- TEACH POPTOLISP

Lisp to Pop Translation

	CLisp	Pop-11
Parentheses	(. . .)	(. . .)
List Brackets	(. . .)	[. . .]
String Quote	". . . "	'. . . '
Word Quote	'*<item>*	"*<item>* "
End Line Comment	;	;;;
Comment Brackets	# |. . . |#	/*. . . */
List construction	'(. . .)	[. . .]
List item insertion	,	^
List Seq. insertion	,@	^^
Procedure Calls	(f x y z)	f(x,y,z)
Operators	(+ a 2 c)	a+2+c
	(+ (∗ a b)(f c))	a∗b+f(c)
Global Variables	(defvar fred 7)	**vars** fred=7;
Assignment to Var.	(setq x (∗ 3 5))	3∗5−>x;

Variables and Procedures

- In Common Lisp the procedure named **foo** has *nothing to do with* the variable named **foo**

  (car '(a b c)) has result A

  (setq car 7) sets *variable* "car" to 7

  (car '(a b c)) still has result A, i.e. built in *procedure* car

  variable car still has value 7

  (+ car 5) has value 12

- Each identifier in Common Lisp can have 2 different "values" at the same time:
  - A procedural(function) value
  - An ordinary value
- In Pop-11 variables only have one value

Procedures and Variables

- Procedure definitions

(defun <*nme*> (<*ps*>) <*expression1*> ⋮ <*expressionN*>)	**define** <*nme*> (<*ps*>); <*expression1*> ⋮ <*expressionN*> **enddefine**;

- Local Variables (inside a procedure)

(let ((x 0) y) <*expressions*>)	**lvars** x=0,y; <*expressions*>

- Updater calls (when updaters exist)

(setf (f x y) v)	v−>f(x,y);
(setf x 3)	3−>x;

Lists

Common Lisp	Pop-11
nil	nil
()	[ ]
(null list)	null(list) OR list==[]
(listp list)	islist(list)
(cons x list)	cons(x,list) OR x::list
(list 1 x ”a string” 0)	[% 1, x, ’a string’, 0 %]
(car list) (first list)	hd(list)
(cdr list) (rest list)	tl(list)

List Processing

- Compound list accessing

Lisp	Pop-11
(car (cdr list)) (cadr list) (second list)	hd(tl(list))
(cdr (cdr list)) (cddr list)	tl(tl(list))
(caddr list)	hd(tl(tl(list)))

- **cadadr** etc. defined for up to 4 or 5 basic operations

- Updating a list

(rplaca list x) (setf (car list) x)	x−>hd(list)
(rplacd list y) (setf (cdr list) y)	y−>tl(list)

- Standard procedures:

(member x list)	member(x,list)
(append list1 list2)	list1<>list2
(length list)	length(list)
(cons x list)	x::list

Quoting Lists

- In Pop-11 the syntax for lists and for procedure calls is different. In Lisp they are the same. How do we specify which we mean?

- Use **quote**

  (f x (g y) z) means **f(x,g(y),z)**

BUT **(quote ((f x (g y) z))** means **[f x [g y] z]**

- The special character "'" is used as a shorthand:

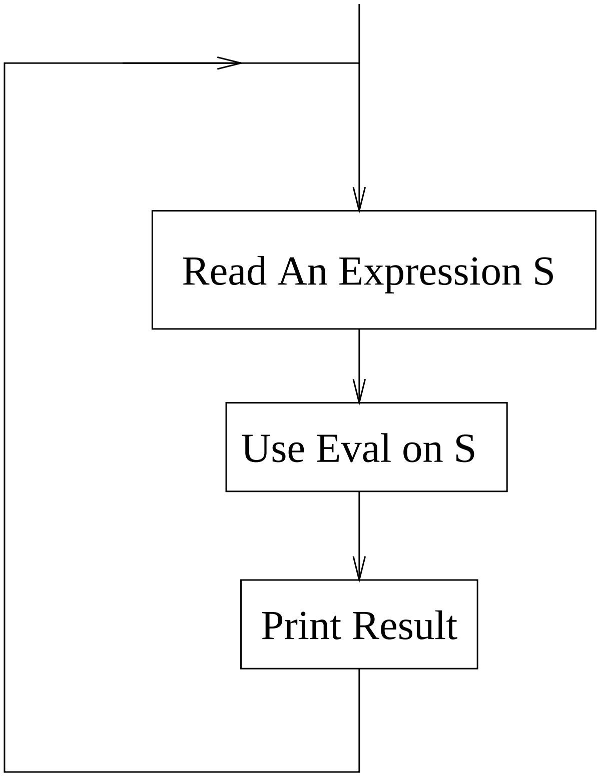
  '(f x y z) is read as **(quote (f x y z))**

- **quote** stops evaluation

  foo means **foo** i.e. the value of **foo**

BUT **'foo** or **(quote foo)** means "foo"

Top-Level Read-Eval-Print Loop



Read An Expression S

Use Eval on S

Print Result

- **Note:** Every Lisp expression has a value

How to Use CLisp

- At Unix prompt:

```
%clisp
```

- Common Lisp will announce itself:

```
Sussex Poplog (Version ...)
Copyright (c) 1982-1995 ...
Common Lisp (Version 2.0)

Setlisp
==
```

This is the Lisp prompt

- Type at prompt

```
== (+ 1 2 3)

   6

==
```

- To leave Common Lisp type:

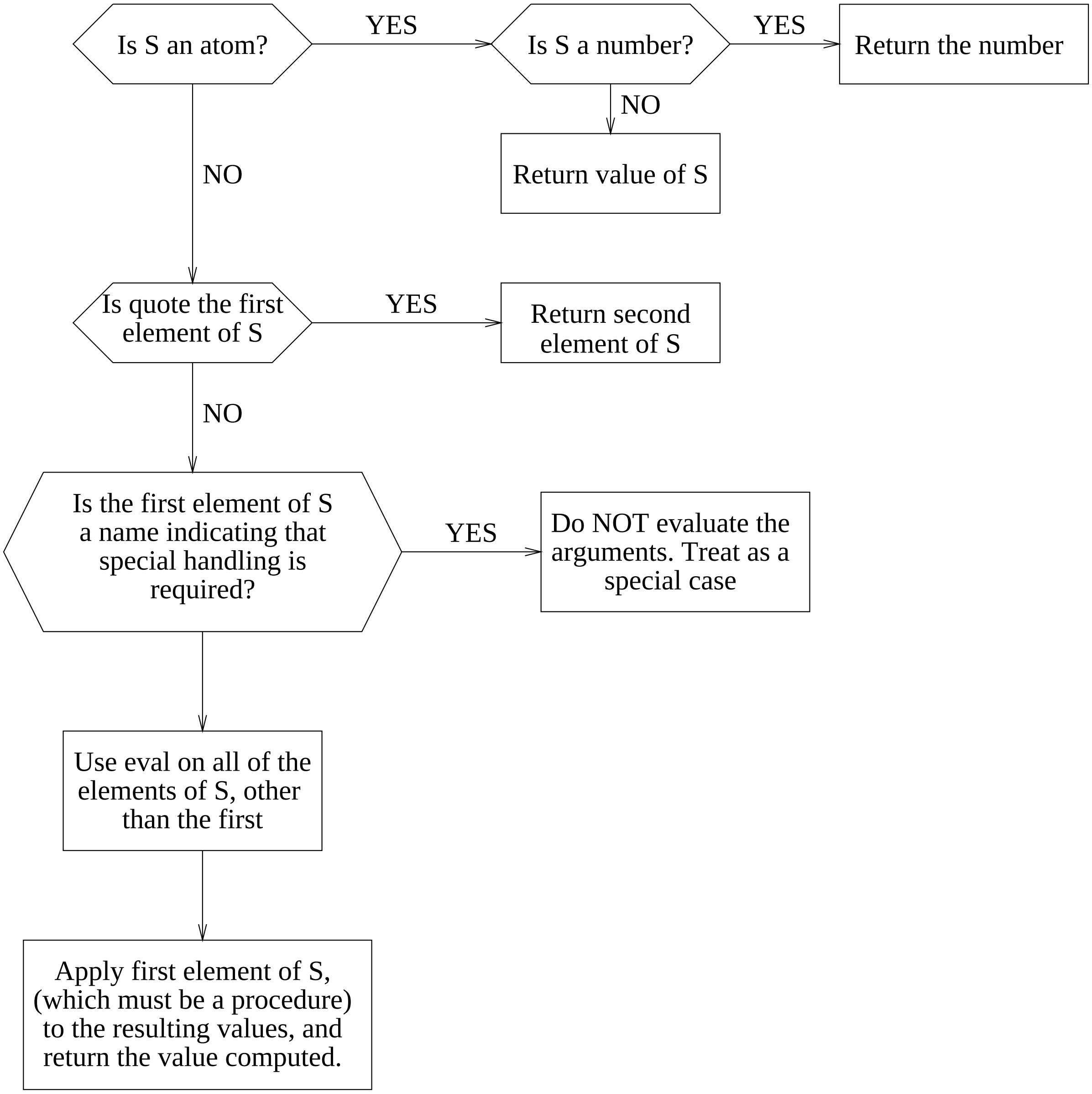
```
== (bye)
```

- To start ved

== ved myfile.lsp *or whatever!*

- Normal ved commands work as normal
- Lisp files should end in **.lsp**
- See **HELP CLISP**

Definition of Eval



Is S an atom?

YES

Is S a number?

YES

Return the number

NO

Return value of S

NO

Is quote the first element of S

YES

Return second element of S

NO

Is the first element of S a name indicating that special handling is required?

YES

Do NOT evaluate the arguments. Treat as a special case

Use eval on all of the elements of S, other than the first

Apply first element of S, (which must be a procedure) to the resulting values, and return the value computed.

- Figure from Winston and Horn
- Assumes quoting done by **(quote ...)**

Logic and Conditionals

Common Lisp	Pop-11
t	true
nil	false
(not <*expr*>)	not(<*expr*>)
(and <*e1*> <*e2*>)	<*e1*> and <*e2*>
(or <*e1*> <*e2*>)	<*e1*> or <*e2*>
(if <*condition*> <*thenexpr*> <*elseexpr*>)	if <*condition*> then <*thenexpr*> else <*elseexpr*> endif;

Predicates

- Type Testing

Common Lisp	Pop-11
(null list)	null(list)
(listp x)	islist(x)
(atom x)	atom(x)
(symbolp x)	isword(x)
(numberp x)	isnumber(x)

- Equality

(eq x y)	x==y
(= x y)	x=y *(numbers only)*
(equal x y)	x=y *("similar things")*

- Arithmetic Comparison

(> x y)	x>y
(< x y)	x<y
(>= x y)	x>=y
(<= x y)	x<=y

Examples

- Append

```
(defun append (lsta lstb)
   (if (null lsta)
         lstb
         (cons (car lsta)
               (append (cdr lsta) lstb)
         )
   )
)
```

- Length

```
(defun length (list)
   (if (null list)
         0
         (+ 1 (length (cdr list)))
   )
)
```

More Conditionals

Common Lisp	Pop-11

(when <*condition*>
<*expression*>
)
	if <*condition*> then
	<*expression*>
	endif;

(unless <*condition*> <*expression*>)	unless <*condition*> then <*expression*> endunless;

(cond (<*condition1*>
<*exprs1*>)
(<*condition2*>
<*exprs2*>)
⋮
(t
<*elseexprs*>)
)
	if <*condition1*>
	then <*exprs1*>
	elseif <*condition2*>
	then <*exprs2*>
	⋮
	else
	<*elseexprs*>
	endif;

Another Example

- A member function

```
(defun mymember(item list)
   (cond ((null list) nil)
         ((eq item (car list)) t)
         (t (mymember item (cdr list)))
   )
)
```

Loops

Common Lisp	Pop-11
(loop <*expressions*>)	repeat <*expressions*> endrepeat

(dotimes (i n)
<*expressions*>
)
	for i from 0 to n-1 do
	<*expressions*>
	endfor;

(dolist (x list)
<*expressions*>
)
	for x in list do
	<*expressions*>
	endfor

- There are also other loops, more akin to those in C e.g. **do**

Yet Another Example

- Intersection

```
(defun intersection (lista listb)
   (let ((ans nil))
      (dolist (x lista)
         (if (member x listb)
            (setq ans (cons x ans))
         )
      )
      ans
   )
)
```

Prog

Common Lisp

```
(prog (x (y 3))
   <exprs1>
foo
   <exprs2>
   (go foo)
   <exprs3>
)
```

Pop-11

```
lvars x,y=3;
   <exprs1>
foo:
   <exprs2>
   goto foo;
   <exprs3>
```

- Progs return **nil** unless use **return**

```
(prog (...)
   ...
   (return 42)
   ...
)
```

- **Note:** In Pop-11 **return** always returns from the enclosing procedure. The Lisp equivalent to this is **return-from**

e.g. (return-from foo (+ n 5))

Specialised Versions of Prog

- Prog1

```
(prog1
   <exp1>
   ⋮
   <expN>
)
```

Executes the sequence of expressions and returns value of first one

- ProgN

```
(progN
   <exp1>
   ⋮
   <expN>
)
```

Executes the sequence of expressions and returns value of last one

Printing and Reading

- Output

Common Lisp	Pop-11
(terpri)	nl(1);
(printc x)	pr(x);
(print x)	nl(1); spr(x);

- Input

(read) returns a Lisp object

e.g. If Lisp reads

(a (b (2 3) c))

it returns a nested list (C.F. **listread** in Pop-11).

e.g. If Lisp reads

”a string”

it returns a string

e.g. If Lisp reads

foo

it returns the symbol ’foo

Anonymous Functions

- In Lisp these are called **lamda** expressions:

Common Lisp	Pop-11
(lambda (<*args*>) <*expressions*>)	**procedure**(<*args*>); <*expressions*> **endprocedure**

- Can pass functions as arguments <u>either</u> by passing the *name*, <u>or</u> by passing an anonymous procedure

e.g (mapcar 'square '(1 2 3))

No direct Pop-11 equivalent

e.g (mapcar (function square) '(1 2 3))

In Pop-11: maplist([1 2 3], square)

e.g. (mapcar #'(lamda (x) (* x x)) '(1 2 3))

In Pop-11:

```
maplist([1 2 3],procedure(x);
                  x*x
              endprocedure)
```

- The following will not work in Common Lisp:

Common Lisp (NOT OK)	Pop-11 (OK)

Common Lisp (NOT OK)

```
(defun apply (f x)
   (f x)
)
```

Pop-11 (OK)

```
define apply(f,x);
   f(x)
enddefine;
```

- We must use **funcall**

e.g.

```
(defun apply (f x)
    (funcall f x)
)
```

Property Lists

- Property lists are lists of the form:

  (colour red size 3 age 2)

- They can be accessed by:

  (getf <*plist*> <*pname*>)
  returns the value

- They can be updated by:

(setf (getf <*plist*> <*pname*>) <*pvalue*>)

- Every symbol has an associated property list, accessed by:

(get <*symbol*> <*pname*>)

- and updated by:

(setf (get <*symbol*> <*pname*>) <*pvalue*>)

An Example

```
(setf (get ’clyde ’species) ’elephant)
(get ’clyde ’species)
```

- These are analogous to properties in Pop-11