Lists and Recursion

CMPU 101 – Problem Solving and Abstraction

Peter Lemieszewski
GNU is an operating system which is 100% free software. It was launched in 1983 by Richard Stallman (rms) and has been developed by many people working together for the sake of freedom of all software users to control their computing. Technically, GNU is generally like Unix. But unlike Unix, GNU gives its users freedom.

The GNU system contains all of the official GNU software packages (which are listed below), and also includes non-GNU free software, notably TeX and the X Window System. Also, the GNU system is not a single static set of programs; users and distributors may select different packages according to their needs and desires. The result is still a variant of the GNU system.

If you're looking for a whole system to install, see our list of GNU/Linux distributions which are entirely free software.

To look for individual free software packages, both GNU and non-GNU, please see the Free Software Directory: a categorized, searchable database of free software. The Directory is actively maintained by the Free Software Foundation and includes links to program home pages where available, as well as entries for all GNU packages. Another list of all GNU packages is below. Free
But what does GNU mean?

What does the G in GNU stand for?

GNU's not Unix

GNU (pronounced as two syllables with a hard g, "ga new") is a recursive acronym standing for "GNU's not Unix". The first goal of the project, initiated for the Free Software Foundation by Richard Stallman, was to produce a fully functional Unix-compatible operating system completely free of copyrighted code.

Jun 18, 2019
But what does recursion mean?

*Recursion* is a programming technique that involves defining a solution or structure using *itself* as part of the definition.

We will revisit recursion again!
Back to lists columns

Columns in a table can contain a mix of different data types, e.g.,

```plaintext
table:
grades
row: 98
row: 56
row: 74
row: "F"
row: "A"
row: "B"
end
```

And so can a list:

```plaintext
[98, 56, 74, "F", "A", "B"]
```
However, we usually find it easier to work with a column where every value is of the same kind.

We can *annotate* the type of data in the column when we make a table:

```plaintext
table:
  col :: Number
  row: 1
  row: 2
  row: 3
end
```

```plaintext
table:
  col :: String
  row: "a"
  row: "b"
  row: "c"
end
```
Back to lists

As we saw with

`string-join` & `string-split` functions, we’ll most often have just one type of data in a list, and we can show that when we write the type annotation for a function:

For example,

- `List<Number>`
  - “list of numbers”

- `List<String>`
  - “list of strings”

[![list: 1, 2, 3]]

[![list: "a", "b", "c"]](list: "a", "b", "c")
In pyret... we can use get...

Much like the rows in a table, the items in a list have (zero based) numeric indices and be accessed via get:
In Pyret... we can use `get`... uh oh.

Much like the rows in a table, the items in a list have (zero based) numeric indices and be accessed via `get`:

```
use context essentials2021

lst = ["a", "b", "c"]

get(lst, 1)
```

The identifier `get` is unbound:

- `get(lst, 1)`

It is used but not previously defined.
In Pyret... we can use `get`... click run first though.

Much like the rows in a table, the items in a list have numeric (zero based) indices and be accessed via `get` as long as we use context essentials2021:

```pyret
use context essentials2021

>>> lst = [list: "a", "b", "c"]

>>> get(lst, 0)

"a"

>>> 
```
List length

```python
>>> lst = ["a", "b", "c"]

The length of a list is always one more than the last item index:

```python
>>> len(lst)
3
```
List member

```python
>>> lst = ["a", "b", "c"]

To check if an item is in a list, we can just ask if the list has it as a member:

```python
>>> member(lst, "c")
true
```
Table functions analogous to List functions

<table>
<thead>
<tr>
<th>Tables</th>
<th>Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform-column</td>
<td>L.map</td>
</tr>
</tbody>
</table>
Table functions analogous to List functions

<table>
<thead>
<tr>
<th>Tables</th>
<th>Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform-column</td>
<td>map</td>
</tr>
<tr>
<td>filter-with</td>
<td>filter</td>
</tr>
</tbody>
</table>
Filter documentation

```
filter :: (f :: (a -> Boolean), lst :: List<a>) -> List<a>
```

Returns the subset of lst for which f(elem) is true

Examples:

```
check:
    fun length-is-one(s :: String) -> Boolean:
        string-length(s) == 1
    end
    filter(length-is-one, [list: "ab", "a", ",", "c"] is [list: "a", "c"]
    filter(is-link, [list: empty, link(1, empty), empty]) is [list: link(1, empty)]
    end
```
List filter example + lambda

```python
>>> lst = ["a", "b", "c"]
>>> filter(
    lam(i): not(i == "a") end,
    lst)
[list: "b", "c"]
```

This is an anonymous (i.e., unnamed) function made using a lambda expression.
Consistently inconsistent pyret functions

One difference to be aware of:

\[
\text{filter-with(} \langle \text{table} \rangle, \langle \text{function} \rangle \rangle
\]

\[
\text{filter(} \langle \text{function} \rangle, \langle \text{list} \rangle \rangle
\]
DIY List functions

Consider: a list of numbers
What do we want: the sum of these numbers
How do we approach this problem?
DIY List functions

Consider: a list of numbers
What do we want: the sum of these numbers
How do we approach this problem?
Similar to how you (hopefully) approached exam problems:
  • Start with a name and write the function shell!

```hs
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
end
```
DIY List functions

Next up: consider testing examples (where...)

- Btw, function “sum” already exists in pyret

```pyret
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
```
DIY List functions: developing test cases before code

Simplest case: an empty list!

- Similar to an empty string: ""
- Corresponds to \[\text{list: }\]

```plaintext
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
  where:
  my-sum([\text{list: }]) is 0 # we could name our empty list and use its name here instead
end
```
DIY List functions: developing test cases before code

Next simplest case: one item in list

•

  • Corresponds to [list: 7]

fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
  where:
    my-sum([list: ]) is 0 # we could name our empty list and use its name here instead
    my-sum([list: 7]) is 7
  end
DIY List functions: developing test cases before code

Next simplest cases: etc.

- Corresponds to [list: 7]

fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
  where:
  my-sum([list: ]) is 0 # we could name our empty list and use its name here instead
  my-sum([list: 7]) is 7
  my-sum([list: 2, 7]) is 9
  my-sum([list: 4, 2, 7]) is 4 + 2 + 7 # math is hard at 3am!
end
Let’s rewrite all of our test case results

In terms of previous results

fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is 0
  my-sum([list: 7]) is 7
  my-sum([list: 2, 7]) is 2 + 7
  my-sum([list: 4, 2, 7]) is 4 + 2 + 7
end
DIY List functions: establishing a pattern

Let’s rewrite all of our test case results

fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"

    ... 

where:
    my-sum([list: ]) is 0
    my-sum([list: 7]) is 7 + 0
    my-sum([list: 2, 7]) is 2 + 7 + 0
    my-sum([list: 4, 2, 7]) is 4 + 2 + 7 + 0

end
DIY List functions: establishing a pattern

Let’s rewrite all of our test case results

fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
  where:
  my-sum([list: ]) is 0
  my-sum([list: 7]) is 7 + my-sum([list: ])
  my-sum([list: 2, 7]) is 2 + my-sum([list: 7])
  my-sum([list: 4, 2, 7]) is 4 + my-sum([list: 2, 7])
end
Before we continue,

We should take a look at:

The Secret Nature of Lists!:
Shorthand list notation

[list: 3, 1, 4] is a lie
List notation in pyret

• Pyret’s *a priori* assumption about lists:
  • There are two ways of making a list.
  • A list is one of either:
    • empty
    • link(⟨item⟩, ⟨list⟩)  # link() will join an item & existing list into a new list.
Implementation of a list

• A list of one item, e.g.,
• [list: "A"],
• is really a link between an item and the empty list:
• link("A", empty)
Implementation of a list

• And so on...
• \([\text{list: } "Z", "A"]\),
• is really a link between an item and the list with one item which itself is a link with one item and the empty list:
• \(\text{link}("Z", (\text{link}("A", \text{empty})))\)
Implementation of a list

- And so on...
- `[list: "Z", "A"],`
- is really a link between an item and the list with one item which itself is a link with one item and the empty list:
  - `link("Z", (link("A", empty)))`

```python
>>> link("Z", (link("A", empty)))
list: "Z", "A"
```
• We now return to our regularly scheduled lecture
From earlier... without the garish colors

Let’s rewrite all of our test case results

fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is 0
  my-sum([list: 7]) is 7 + my-sum([list: ])
  my-sum([list: 2, 7]) is 2 + my-sum([list: 7])
  my-sum([list: 4, 2, 7]) is 4 + my-sum([list: 2, 7])
end
Writing the my-sum function

To write our function, we need to use the true form of a list and think *recursively*.

```haskell
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"
    #thinking recursively...

where:
    my-sum([list: ]) is 0
    my-sum([list: 7]) is 7 + my-sum([list: ])
    my-sum([list: 2, 7]) is 2 + my-sum([list: 7])
    my-sum([list: 4, 2, 7]) is 4 + my-sum([list: 2, 7])
end
```
Writing the my-sum function, recursively (1)

fun **my-sum**(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  # our base “case” is when our list is empty: ([list: ]) is 0
  # the next case is when our list is NOT empty

where:

- my-sum([list: ]) is 0
- my-sum([list: 7]) is 7 + my-sum([list: ])
- my-sum([list: 2, 7]) is 2 + my-sum([list: 7])
- my-sum([list: 4, 2, 7]) is 4 + my-sum([list: 2, 7])

end
We’ll refer to this as The Base Case

fun my-sum(lst :: List<Number>) -> Number:

doc: "Return the sum of the numbers in the list"

cases (List) lst: | empty => # the answer when list is empty!

0

# the next case

where:

my-sum([list: ]) is 0

my-sum([list: 7]) is 7 + my-sum([list: ])

my-sum([list: 2, 7]) is 2 + my-sum([list: 7])

my-sum([list: 4, 2, 7]) is 4 + my-sum([list: 2, 7])
end
We’ll refer to this as The **Recursive Case**

```plaintext
fun my-sum(lst :: List<Number>) -> Number:

doc: "Return the sum of the numbers in the list"

cases (List) lst:
  | empty => 0
  | link(f, r) => f + my-sum(r) # covers all the other tests in where

where:

my-sum([list: ]) is 0

my-sum([list: 7]) is 7 + my-sum([list: ])
my-sum([list: 2, 7]) is 2 + my-sum([list: 7])
my-sum([list: 4, 2, 7]) is 4 + my-sum([list: 2, 7])

end
```
How does this function get evaluated?

When we call this function, it evaluates as:

- \( \text{my-sum(link(3, link(1, link(4, empty))))} \)
- \( 3 + \text{my-sum(link(1, link(4, empty)))} \)
- \( 3 + 1 + \text{my-sum(link(4, empty))} \)
- \( 3 + 1 + 4 + \text{my-sum(empty)} \)
- \( 3 + 1 + 4 + 0 \)
When is a recursive solution appropriate?

Any time a problem is structured such that
• the solution on larger inputs can be built from the solution on smaller inputs, then
• recursion is appropriate.
The two cases we need to solve

All recursive functions have these two parts:

* **Base case(s):**
  What’s the simplest case to solve?

* **Recursive case(s):**
  What’s the relationship between the current case and the answer to a slightly smaller case?

You should be calling the function you’re defining here; this is referred to as a recursive call.

Each time you make a recursive call, you must make the input smaller.

  Otherwise, we would have a “GNU” case (i.e. endless recursion)!

If your input is a list, you do this by passing the rest of the list to the recursive call.
Splitting up a list recursively: First and Rest

```python
>>> lst = ["item 1", "and", "so", "on"]
>>> lst.first
"item 1"
>>> lst.rest
["and", "so", "on"]
```
First/Rest in my-sum

\[
\text{link}(f, r) \Rightarrow f + \text{my-sum}(r)
\]

- first of the list is... \( f \)
- rest of the list is... \( \text{my-sum}(r) \)
What if...

... we made a recursive call on the original input list?

\[
\text{link}(f, r) \Rightarrow f + \text{my-sum}(\text{lst})
\]

- first of the list is... \(f\)
- rest of the list is... \(\text{my-sum}(\text{lst})\)
Let’s try writing another recursive function

Given: a list of numbers…

The function `any-below-10` should return `true` if any member of the list is less than 10 and `false` otherwise.
#Start with the test cases first!

fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"
  ...
  where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or (1 < 10) or (4 < 10)
  any-below-10([list: 1, 4]) is (1 < 10) or (4 < 10)
  any-below-10([list: 4]) is (4 < 10)
  any-below-10([list: ]) is ...
end
fun any_below-10(lst :: List<Number>) -> Boolean:
    doc: "Return true if any number in the list is less than 10"
    ...
where:
    any_below-10([list: 3, 1, 4]) is (3 < 10) or (1 < 10) or (4 < 10)
    any_below-10([list: 1, 4]) is (1 < 10) or (4 < 10)
    any_below-10([list: 4]) is (4 < 10)
    any_below-10([list: ]) is false
end
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"
  ...
  where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or any-below-10([list: 1, 4])
  any-below-10([list: 1, 4]) is (1 < 10) or any-below-10([list: 4])
  any-below-10([list: 4]) is (4 < 10) or any-below-10([list: ])
  any-below-10([list: ]) is false
end
Writing any-below-10: lasty, the function itself

fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"
  cases (List) lst:
    | empty => false
    | link(f, r) => (f < 10) or any-below-10(r)

where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or any-below-10([list: 1, 4])
  any-below-10([list: 1, 4]) is (1 < 10) or any-below-10([list: 4])
  any-below-10([list: 4]) is (4 < 10) or any-below-10([list: ])
  any-below-10([list: ]) is false
end
Writing a Recursive Predicate

• Now that we’ve seen how to write any-below-10, we can use the same pattern to implement a higher-order function where we can ask if any item in a list satisfies some predicate.
  • “Some predicate”: meaning some kind of “generalized or, helper, function”
fun my-any(fn :: Function, lst :: List) -> Boolean:
  doc: "Return true if the function fn is true for any item in the given list."
  cases (List) lst:
    | empty => false
    | link(f, r) => fn(f) or my-any(fn, r)
end
end
Writing my-all

fun my-all(fn :: Function, lst :: List) -> Boolean:
  doc: "Return true if the function fn is true for every item in the given list."
  cases (List) lst:
      | empty => true
      | link(f, r) => fn(f) and my-all(fn, rst)
  end
end
Let’s try some practice examples together

BTW This stuff can be *adjective*!

```
adjectives = [list: “difficult", “funky""]
```
fun list-len(lst :: List) -> Number:
    doc: "Compute the length of a list"
    cases (List) lst:
        | empty => 0
        | link(f, r) => 1 + list-len(____)
    end
end
end
fun list-len(lst :: List) -> Number:
  doc: "Compute the length of a list"
  cases (List) lst:
    | empty => 0
    | link(f, r) => 1 + list-len(r)
  end
end
fun list-product(lst :: List<Number>) -> Number:
   doc: "Compute the product of all the numbers in lst"
   cases (List) lst:
      | empty => 1
      | link(f, r) => ____ * list-product(r)
   end
end

Practice Makes _____
fun list-product(lst :: List<Number>) -> Number:
  doc: "Compute the product of all the numbers in lst"
  cases (List) lst:
    | empty => 1
    | link(f, r) => f * list-product(r)
  end
end
fun is-member(lst :: List, item) -> Boolean:
  doc: "Return true if item is a member of lst"
  cases (List) lst:
    | empty => ______
    | link(f, r) =>
      (f == ______) or (is-member(______, ______))
  end
end

Practice Makes ______
fun is-member(lst :: List, item) -> Boolean:
  doc: "Return true if item is a member of lst"
  cases (List) lst:
    | empty => false
    | link(f, r) =>
      (f == item) or (is-member(r, item)
  end
end
Link to code

• https://code.pyret.org/editor#share=11g-ulsllopYJlZUctfv9wIpDN9rTIVFW&v=31c9aaf
Acknowledgements

• This lecture incorporates material from:
• Kathi Fisler, Brown University,
• Marc Smith, Vassar College
• And, Jonathan Gordon, Vassar College