Ling 5801: Lecture Notes 4
From Unix Scripts to Programs

Unlike simple chains of unix commands, programs are *recursive* (nested).

We will define programming languages using a grammar (as we later define natural languages).

Contents

4.1 Programs

Programs are sequences of characters made up of *recursive* (nested) sub-types.

Some common types:

- $\langle$program$\rangle$

  this is a top-level type for an entire program (like a document)
- $\langle$stmt$\rangle$

  a statement is a sequence of characters that describes a desired action (like a sentence)
- $\langle\alpha$-expr$\rangle$

  an expression is a string that describes a value (like a phrase), e.g. $\langle$num-expr$\rangle$ describes a number

Typed character sequences recursively decompose into sub-sequencess of other types.

For example, here is a simple subset of the programming language Python

(for now, interpret '$\rightarrow$' as 'may consist of'):

1. $\langle$program$\rangle$ $\rightarrow$ $\langle$stmt$\rangle$

   a program may consist of a single statement

2. ⟨program⟩ → ⟨stmt⟩ NEWLINE ⟨program⟩

   a program may consist of a sequence of delimited statements

   (Python pays attention to indentation, so top-level statements must begin at left margin!)

We'll also define a statement for printing to standard output (there will be more):

3. ⟨stmt⟩ → `print ( `⟨α-expr⟩` )`

   a statement may be a print command followed by any type of argument expression

We'll also define a type of expression for 'strings' of characters (there will be more):

4. ⟨string-expr⟩ → `'` ⟨`[A-Za-z0-9.,!? \n]*`⟩ `'`

   a string expression may consist of a bunch of characters between quotes

   (`\n` is a new-line character in a string; like typing 'carriage return' or 'enter')

5. ⟨string-expr⟩ → ⟨string-expr⟩ `+` ⟨string-expr⟩

   a string expression may consist of two string expressions concatenated together

Now we can write a simple program:

(type '`python`' in unix Terminal window to enter interpreter, then type the program)

(you can also edit in TextEdit, say '`myprog.py`', then run using '`python myprog.py`')

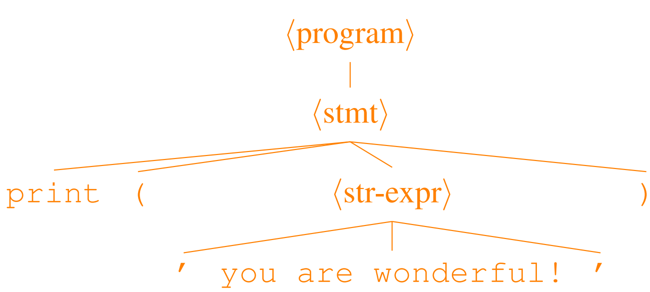
```
print ( 'you are wonderful!' )
```

this will print:

```
you are wonderful!
```

4.2 Recursive types within a program

The nested or 'recursive' types in a program can be drawn as a tree:



4.3 Numerical expressions

We can also print other things:

1. $\langle$num-expr$\rangle$ $\rightarrow$ $\langle$[0-9]+$\rangle$

   a number expression may consist of a bunch of numerals (denoted using regexp)

2. $\langle$num-expr$\rangle$ $\rightarrow$ $\langle$num-expr$\rangle$ `+` $\langle$num-expr$\rangle$

   a number expression may be an addition of two number expressions (result is the sum)

3. $\langle$num-expr$\rangle$ $\rightarrow$ $\langle$num-expr$\rangle$ `-` $\langle$num-expr$\rangle$
4. $\langle$num-expr$\rangle$ $\rightarrow$ $\langle$num-expr$\rangle$ `*` $\langle$num-expr$\rangle$
5. $\langle$num-expr$\rangle$ $\rightarrow$ $\langle$num-expr$\rangle$ `/` $\langle$num-expr$\rangle$

   same for other operators

6. $\langle\alpha$-expr$\rangle$ $\rightarrow$ `(` $\langle\alpha$-expr$\rangle$ `)`

   a number (or any other) expression may be surrounded by parentheses

7. $\langle$num-expr$\rangle$ $\rightarrow$ `int(` $\langle$string-expr$\rangle$ `)`

   a string can be converted into a number expression, e.g. for reading

And here are some things we can do with them:

8. $\langle$string-expr$\rangle$ $\rightarrow$ `str(` $\langle$num-expr$\rangle$ `)`

   a number can be converted into a string expression, e.g. for printing

Now we can use Python as a calculator:

```
print ( (2+4)/3 )
```

will print:

```
2
```

Practice:

Draw the above program as a tree.

4.4 Boolean expressions

Logical inference is handled using Boolean expressions, which are `True` or `False`:

1. $\langle$bool-expr$\rangle$ $\rightarrow$ `True`
2. $\langle$bool-expr$\rangle$ $\rightarrow$ `False`

   a Boolean expression may be a capitalized constant true/false value

3. $\langle$bool-expr$\rangle$ $\rightarrow$ $\langle$bool-expr$\rangle$ `and` $\langle$bool-expr$\rangle$

   a Boolean expression may be a conjunction of two Boolean exprs (true if both true)

4. ⟨bool-expr⟩ → ⟨bool-expr⟩ `or` ⟨bool-expr⟩

   a Boolean expression may be a disjunction of two Boolean exprs (true if either true)

5. ⟨bool-expr⟩ → `not` ⟨bool-expr⟩

   a Boolean expression may be a negation of another Boolean expr (true if subexpr false)

6. ⟨bool-expr⟩ → ⟨num-expr⟩ `>` ⟨num-expr⟩

7. ⟨bool-expr⟩ → ⟨num-expr⟩ `<` ⟨num-expr⟩

8. ⟨bool-expr⟩ → ⟨α-expr⟩ `==` ⟨α-expr⟩

   a Boolean expression may be a (greater than / less than / equality) test on number exprs

   (NOTE: you must use double-equals here! single equals is something else!)

Now we can use Python as a math checker:

```
print ( (2+4)/3 == 2 )
```

will print:

```
True
```

4.5 Conditionals

Programs behavior can depend on Boolean conditions:

1. ⟨stmt⟩ → `if` ⟨bool-expr⟩ `:` NEWLINE ⟨suite⟩

   perform ⟨suite⟩ if ⟨bool-expr⟩ is true

2. ⟨stmt⟩ → `if` ⟨bool-expr⟩ `:` NEWLINE ⟨suite⟩ NEWLINE `else` `:` NEWLINE ⟨suite⟩

   perform first ⟨suite⟩ if ⟨bool-expr⟩ is true, otherwise perform second ⟨suite⟩

where a suite is an indented sub-program, defined in terms of modifications to the margin:

3. ⟨suite⟩ → INDENT ⟨program⟩ DEDENT

   INDENT : add spaces to old margin get new margin;

   DEDENT : subtract spaces to return to previous margin

(interpreter may require entering an empty line to show you're done with the indented part)

For example:

```
if 2<3:
  print ( 'Computer is working.'  )
else:
  print ( 'Computer is broken.'  )
```

will print:

```
Computer is working.
```

4.6 Variables

In addition to printing, we can also store values in variables:

1. ⟨stmt⟩ → ⟨α-var⟩ = ⟨α-expr⟩

   store ⟨α-expr⟩ in a variable (memory location) named ⟨α-var⟩

2. ⟨α-expr⟩ → ⟨α-var⟩

   a number expression may be a number variable (evaluates to contents of variable)

3. ⟨α-var⟩ → ⟨[A-Za-z_][A-Za-z_0-9]*⟩

   a variable may consist of a bunch of letters or numbers

For example:

```
x = 3
x = x - 1
print ( x )
```

will print:

```
2
```

4.7 Lists

Variables can store lists of values (including lists of lists):

1. ⟨α-list-expr⟩ → [ ]
2. ⟨α-list-expr⟩ → [⟨α-list-element-seq⟩]
3. ⟨α-list-element-seq⟩ → ⟨α-expr⟩
4. ⟨α-list-element-seq⟩ → ⟨α-expr⟩ , ⟨α-list-element-seq⟩
5. ⟨α-list-expr⟩ → range (⟨num-expr⟩, ⟨num-expr⟩)

   lists may contain nothing / expressions / numbers from first ⟨num-expr⟩ to second ⟨num-expr⟩

6. ⟨α-list-expr⟩ → ⟨α-list-expr⟩ + ⟨α-list-expr⟩

   lists can be combined by concatenation

And here are some things we can do with them:

7. ⟨α-var⟩ → ⟨α-list-var⟩ `[` ⟨num-expr⟩ `]`

   list elements can be indexed by number

8. ⟨num-expr⟩ → `len(` ⟨α-list-expr⟩ `)`

   a number expression can be the length of a list ⟨α-list-expr⟩

9. ⟨bool-expr⟩ → ⟨α-expr⟩ `in` ⟨α-list-expr⟩

   a boolean can indicate true if ⟨α-expr⟩ is in ⟨α-list-expr⟩, false otherwise

10. ⟨bool-expr⟩ → ⟨α-expr⟩ `not in` ⟨α-list-expr⟩

    a boolean can indicate false if ⟨α-expr⟩ is in ⟨α-list-expr⟩, true otherwise

For example, these rules can recursively define a list of list of numbers:

```
A = [ [ 17, 14 ], [ 21 ] ]
print ( A[0][1] )
```

will print:

```
14
```

Practice:

Write an expression that would output the '`21`' from list `A`, above.

4.8 Loops

Programs behavior can repeat (depending on Boolean conditions):

1. ⟨stmt⟩ → `while` ⟨bool-expr⟩ `:` NEWLINE ⟨suite⟩

   repeat ⟨suite⟩ as long as ⟨bool-expr⟩ is true

2. ⟨stmt⟩ → `for` ⟨α-var⟩ `in` ⟨α-list-expr⟩ `:` NEWLINE ⟨suite⟩

   do ⟨suite⟩ for each value in ⟨α-list-expr⟩, assigned to ⟨α-var⟩

For example:

```
for x in range(1,5):
  print ( x )
```

will print:

```
1
2
3
4
```

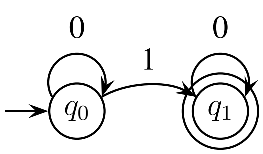
Practice:

Write a program to count to 100 by 3's:

```
3
6
9
12
⋮
```

4.9 Implementation of 'pet language' FSA

Sample Python program implementing FSA:



(text after '#' are treated as comments and ignored)

```
Q=[0, 1]          # set of states (0=ok, 1=hungry)
X=[0, 1]          # set of observation values (0=burble, 1=whiffle)
S=[True, False]   # set of start states (start off fed: true=member, false=not)
F=[False, True]   # set of final states (alert when hungry: true=member, false=not)

# initialize model as list of lists of truth values (think of as a 3-D array)
M=[[[False,False],[False,False]],
   [[False,False],[False,False]]]

M[0][0][0]=True   # model:  'ok'     state on 'burble'  input goes to 'ok'     state
M[0][1][1]=True   #         'ok'     state on 'whiffle' input goes to 'hungry' state
M[1][0][1]=True   #         'hungry' state on 'burble'  input goes to 'hungry' state

Input=[0,1,0]     # input sequence: burble whiffle burble
T=3               # input length

# initialize table of values over time (a 2-D array)
V=[[False,False],[False,False],[False,False],[False,False]]

# initialize first time step with initial state values
for q in Q:
    V[0][q]=S[q]

# compute possible states q in V at each time step t based on possible
# states qP at previous time step t-1 and allowable transitions in M
for t in range(1,T+1):
    for qP in Q:
        for q in Q:
```

```
            V[t][q] = V[t][q] or (V[t-1][qP] and M[qP][Input[t-1]][q])

# if possible to be in any final state at end, accept
for q in Q:
    if ( V[T][q] and F[q] ):
        print ( 'yes' )
```

Practice:

Step through the above code.