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).

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

   Some common types:
   
   - ⟨**program**⟩
     this is a top-level type for an entire program
   - ⟨**stmt-seq**⟩
     a sequence of statements has its own type
   - ⟨**stmt**⟩
     a statement is a sequence of characters that describes a desired action
   - ⟨**delim-stmt**⟩
     a delimited statement is a statement followed by a newline character
   - ⟨**α-exp**⟩
     an expression is a string that describes a value, e.g. ⟨**num-exp**⟩ describes a number

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

   For example, here is a simple subset of the programming language Python
   (for now, interpret ‘→’ as ‘may consist of’):

   (a) ⟨**program**⟩ → ⟨**stmt-seq**⟩
       a program may consist of a sequence of ‘statements’ (which are actually imperative)

   (b) ⟨**stmt-seq**⟩ → ⟨**delim-stmt**⟩ ⟨**stmt-seq**⟩
       a statement sequence may be a delimited statement followed by a statement sequence

   (c) ⟨**stmt-seq**⟩ → ε
       a sequence of statements may also be empty — so an empty file is a valid program!

   (d) ⟨**delim-stmt**⟩ → ⟨**stmt**⟩ [NEWLINE]
       a delimited statement may be a statement followed by a new line and spaces
       (Python pays attention to indentation, so top-level statements must begin at left margin!)

   (e) ⟨**stmt**⟩ → print ( ⟨**α-exp**⟩ )
       a statement may be a print command followed by any type of argument expression

   (f) ⟨**string-exp**⟩ → ‘ ⟨[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’)

1
(g) \( (\text{string-expr}) \rightarrow (\text{string-expr}) + (\text{string-expr}) \)
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`)

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

this will print:

```
you are wonderful!
```

2. Recursive types within a program:

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

```
(program)
    |
  (stmt-seq)
    |
  (stmt-seq)
    |
  (delim-stmt)
    |
  (stmt)
    |
  (str-expr)
    |
  'you are wonderful!'
```

3. Numerical expressions:

We can also print other things:

(a) \( (\text{num-expr}) \rightarrow [0-9]+ \)
a number expression may consist of a bunch of numerals (denoted using regexp)

(b) \( (\text{num-expr}) \rightarrow (\text{num-expr}) + (\text{num-expr}) \)
a number expression may be an addition of two number expressions (result is the sum)

(c) \( (\text{num-expr}) \rightarrow (\text{num-expr}) - (\text{num-expr}) \)

(d) \( (\text{num-expr}) \rightarrow (\text{num-expr}) * (\text{num-expr}) \)

(e) \( (\text{num-expr}) \rightarrow (\text{num-expr}) / (\text{num-expr}) \)

same for other operators

(f) \( (\text{num-expr}) \rightarrow ( (\text{num-expr}) ) \)
a number expression may be surrounded by parentheses
(g) \( \langle \text{string-expr} \rangle \rightarrow \text{str} ( \langle \text{num-expr} \rangle ) \)  
a number can be converted into a string expression, e.g. for printing

(h) \( \langle \text{num-expr} \rangle \rightarrow \text{int} ( \langle \text{string-expr} \rangle ) \)  
a string can be converted into a number expression, e.g. for reading

now we can use Python as a calculator:

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

will print:

```
2
```

4. **Practice:**

Draw the above program as a tree.

5. **Boolean expressions:**

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

(a) \( \langle \text{bool-expr} \rangle \rightarrow \text{True} \)
(b) \( \langle \text{bool-expr} \rangle \rightarrow \text{False} \)

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

(c) \( \langle \text{bool-expr} \rangle \rightarrow \langle \text{bool-expr} \rangle \text{ and } \langle \text{bool-expr} \rangle \)

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

(d) \( \langle \text{bool-expr} \rangle \rightarrow \langle \text{bool-expr} \rangle \text{ or } \langle \text{bool-expr} \rangle \)

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

(e) \( \langle \text{bool-expr} \rangle \rightarrow \text{not} \langle \text{bool-expr} \rangle \)

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

(f) \( \langle \text{bool-expr} \rangle \rightarrow ( \langle \text{bool-expr} \rangle ) \)

a Boolean expression may be surrounded by parentheses

(g) \( \langle \text{bool-expr} \rangle \rightarrow \langle \text{num-expr} \rangle > \langle \text{num-expr} \rangle \)

(h) \( \langle \text{bool-expr} \rangle \rightarrow \langle \text{num-expr} \rangle < \langle \text{num-expr} \rangle \)

(i) \( \langle \text{bool-expr} \rangle \rightarrow \langle \text{num-expr} \rangle == \langle \text{num-expr} \rangle \)

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:

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

now we can use Python as a math checker:
will print:

```
True
```

6. Conditionals:

Programs behavior can depend on Boolean conditions:

(a) `(delim-stmt) → if ⟨bool-expr⟩ : NEWLINE ⟨suite⟩` perform ⟨suite⟩ if ⟨bool-expr⟩ is true

(b) `(delim-stmt) → if ⟨bool-expr⟩ : NEWLINE ⟨suite1⟩ else : NEWLINE ⟨suite2⟩` perform ⟨suite1⟩ if ⟨bool-expr⟩ is true, otherwise perform ⟨suite2⟩

where a suite is defined in terms of modifications to the margin:

(c) `(suite) → INDENT ⟨stmt-seq⟩ DEDENT` `INDENT`: add spaces to old margin get new margin, tab to resulting margin; `DEDENT`: subtract spaces to return to previous margin, tab to resulting margin

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

For example:

```python
if 2<3:
    print ( 'good, numbers are in proper order.' )
else:
    print ( 'uh-oh, you must be dreaming.' )
```

will print:

```
good, numbers are in proper order.
```

7. Variables:

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

(a) `(stmt) → ⟨α-var⟩ = ⟨α-expr⟩` store ⟨α-expr⟩ in a variable (memory location) named ⟨α-var⟩

(b) `(α-expr) → ⟨α-var⟩` a number expression may be a number variable (evaluates to contents of variable)

(c) `(α-var) → ([A-Za-z][A-Za-z0-9]*)` a variable may consist of a bunch of letters or numbers

For example:

```python
x = 3
x = x - 1
print ( x )
```
will print:

2
8. Lists:

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

(a) \( \langle \alpha \text{-list-expr} \rangle \rightarrow [ ] \)
(b) \( \langle \alpha \text{-list-expr} \rangle \rightarrow [ \langle \alpha \text{-list-element-seq} \rangle ] \)
(c) \( \langle \alpha \text{-list-element-seq} \rangle \rightarrow \langle \alpha \text{-expr} \rangle \)
(d) \( \langle \alpha \text{-list-element-seq} \rangle \rightarrow \langle \alpha \text{-expr} \rangle , \langle \alpha \text{-list-element-seq} \rangle \)
(e) \( \langle \alpha \text{-list-expr} \rangle \rightarrow \text{range} (\langle \text{num-expr1} \rangle, \langle \text{num-expr2} \rangle) \)
   lists may contain nothing / expressions / numbers from \( \langle \text{num-expr1} \rangle \) to \( \langle \text{num-expr2} \rangle \)
(f) \( \langle \alpha \text{-list-expr} \rangle \rightarrow \langle \alpha \text{-list-expr} \rangle + \langle \alpha \text{-list-expr} \rangle \)
   lists can be combined by concatenation
(g) \( \langle \alpha \text{-var} \rangle \rightarrow \langle \alpha \text{-list-var} \rangle [ \langle \text{num-expr} \rangle ] \)
   list elements can be indexed by number
(h) \( \langle \text{num-expr} \rangle \rightarrow \text{len} (\langle \alpha \text{-list-expr} \rangle ) \)
   a number expression can be the length of a list \( \langle \alpha \text{-list-expr} \rangle \)
(i) \( \langle \text{bool-expr} \rangle \rightarrow \langle \alpha \text{-expr} \rangle \text{ in } \langle \alpha \text{-list-expr} \rangle \)
   a boolean can indicate true if \( \langle \alpha \text{-expr} \rangle \) is in \( \langle \alpha \text{-list-expr} \rangle \), false otherwise
(j) \( \langle \text{bool-expr} \rangle \rightarrow \langle \alpha \text{-expr} \rangle \text{ not in } \langle \alpha \text{-list-expr} \rangle \)
   a boolean can indicate false if \( \langle \alpha \text{-expr} \rangle \) is in \( \langle \alpha \text{-list-expr} \rangle \), 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

9. Practice:

Write an expression that would output the ‘21’ from list \( A \), above.

10. Loops:

Programs behavior can repeat (depending on Boolean conditions):

(a) \( \langle \text{delim-stmt} \rangle \rightarrow \text{while } \langle \text{bool-expr} \rangle : \text{NEWLINE} \langle \text{suite} \rangle \)
    repeat \( \langle \text{suite} \rangle \) as long as \( \langle \text{bool-expr} \rangle \) is true
(b) \( \langle \text{delim-stmt} \rangle \rightarrow \text{for } \langle \alpha \text{-var} \rangle \text{ in } \langle \alpha \text{-list-expr} \rangle : \text{NEWLINE} \langle \text{suite} \rangle \)
    do \( \langle \text{suite} \rangle \) for each value in \( \langle \alpha \text{-list-expr} \rangle \), assigned to \( \langle \alpha \text{-var} \rangle \)
For example:

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

will print:

1
2
3
4

11. Practice:

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

3
6
9
12
...

12. Implementation of ‘pet language’ FSA

Sample Python program implementing FSA:

(Q=
X=
S=
F=
 # 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
M[0][1][1]=True
M[1][0][1]=True

Input=[0, 1, 0]  # input sequence
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' )

13. Practice:
   Step through the above code.