

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	1
4.2	Recursive types within a program	2
4.3	Numerical expressions	2
4.4	Boolean expressions	3
4.5	Conditionals	4
4.6	Variables	4
4.7	Lists	5
4.8	Loops	6
4.9	Implementation of ‘pet language’ FSA	6

4.1 Programs

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 (like a document)
- `<stmt>`
a statement is a sequence of characters that describes a desired action (like a sentence)
- `< α -expr>`
an expression is a string that describes a value (like a phrase), e.g. `<num-expr>` 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 ‘ \rightarrow ’ as ‘may consist of’):

1. `<program>` \rightarrow `<stmt>`
a program may consist of a single statement
2. `<program>` \rightarrow `<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. $\langle \text{stmt} \rangle \rightarrow \text{print} (\langle \alpha\text{-expr} \rangle)$

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. $\langle \text{string-expr} \rangle \rightarrow ' \langle [A-Za-z0-9.,!?\ \backslash n]^* \rangle '$

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. $\langle \text{string-expr} \rangle \rightarrow \langle \text{string-expr} \rangle + \langle \text{string-expr} \rangle$

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')

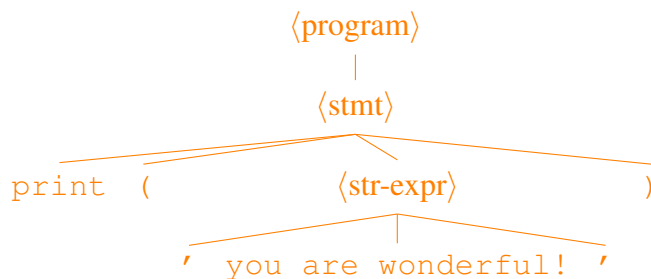
```
print ( 'you are wonderful!' )
```

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. $\langle \text{num-expr} \rangle \rightarrow \langle [0-9]^+ \rangle$

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

2. $\langle \text{num-expr} \rangle \rightarrow \langle \text{num-expr} \rangle + \langle \text{num-expr} \rangle$

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

3. $\langle \text{num-expr} \rangle \rightarrow \langle \text{num-expr} \rangle - \langle \text{num-expr} \rangle$

4. $\langle \text{num-expr} \rangle \rightarrow \langle \text{num-expr} \rangle * \langle \text{num-expr} \rangle$

5. $\langle \text{num-expr} \rangle \rightarrow \langle \text{num-expr} \rangle / \langle \text{num-expr} \rangle$

same for other operators

6. $\langle \alpha\text{-expr} \rangle \rightarrow (\langle \alpha\text{-expr} \rangle)$

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

7. $\langle \text{num-expr} \rangle \rightarrow \text{int} (\langle \text{string-expr} \rangle)$

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

And here are some things we can do with them:

8. $\langle \text{string-expr} \rangle \rightarrow \text{str} (\langle \text{num-expr} \rangle)$

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

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. $\langle \text{bool-expr} \rangle \rightarrow \text{True}$

2. $\langle \text{bool-expr} \rangle \rightarrow \text{False}$

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

3. $\langle \text{bool-expr} \rangle \rightarrow \langle \text{bool-expr} \rangle \text{ and } \langle \text{bool-expr} \rangle$

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

4. $\langle \text{bool-expr} \rangle \rightarrow \langle \text{bool-expr} \rangle \text{ or } \langle \text{bool-expr} \rangle$

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

5. $\langle \text{bool-expr} \rangle \rightarrow \text{not } \langle \text{bool-expr} \rangle$

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

6. $\langle \text{bool-expr} \rangle \rightarrow \langle \text{num-expr} \rangle > \langle \text{num-expr} \rangle$

7. $\langle \text{bool-expr} \rangle \rightarrow \langle \text{num-expr} \rangle < \langle \text{num-expr} \rangle$

8. $\langle \text{bool-expr} \rangle \rightarrow \langle \alpha\text{-expr} \rangle == \langle \alpha\text{-expr} \rangle$

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:

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

will print:

```
True
```

4.5 Conditionals

Programs behavior can depend on Boolean conditions:

1. $\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{bool-expr} \rangle : \boxed{\text{NEWLINE}} \langle \text{suite} \rangle$

perform $\langle \text{suite} \rangle$ if $\langle \text{bool-expr} \rangle$ is true

2. $\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{bool-expr} \rangle : \boxed{\text{NEWLINE}} \langle \text{suite} \rangle \boxed{\text{NEWLINE}} \text{else} : \boxed{\text{NEWLINE}} \langle \text{suite} \rangle$

perform first $\langle \text{suite} \rangle$ if $\langle \text{bool-expr} \rangle$ is true, otherwise perform second $\langle \text{suite} \rangle$

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

3. $\langle \text{suite} \rangle \rightarrow \boxed{\text{INDENT}} \langle \text{program} \rangle \boxed{\text{DEDENT}}$

$\boxed{\text{INDENT}}$: add spaces to old margin get new margin;

$\boxed{\text{DEDENT}}$: 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. $\langle \text{stmt} \rangle \rightarrow \langle \alpha\text{-var} \rangle = \langle \alpha\text{-expr} \rangle$
store $\langle \alpha\text{-expr} \rangle$ in a variable (memory location) named $\langle \alpha\text{-var} \rangle$
2. $\langle \alpha\text{-expr} \rangle \rightarrow \langle \alpha\text{-var} \rangle$
a number expression may be a number variable (evaluates to contents of variable)
3. $\langle \alpha\text{-var} \rangle \rightarrow \langle [A-Za-z_][A-Za-z_0-9]^* \rangle$
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. $\langle \alpha\text{-list-expr} \rangle \rightarrow []$
2. $\langle \alpha\text{-list-expr} \rangle \rightarrow [\langle \alpha\text{-list-element-seq} \rangle]$
3. $\langle \alpha\text{-list-element-seq} \rangle \rightarrow \langle \alpha\text{-expr} \rangle$
4. $\langle \alpha\text{-list-element-seq} \rangle \rightarrow \langle \alpha\text{-expr} \rangle , \langle \alpha\text{-list-element-seq} \rangle$
5. $\langle \alpha\text{-list-expr} \rangle \rightarrow \text{range} (\langle \text{num-expr} \rangle , \langle \text{num-expr} \rangle)$
lists may contain nothing / expressions / numbers from first $\langle \text{num-expr} \rangle$ to second $\langle \text{num-expr} \rangle$
6. $\langle \alpha\text{-list-expr} \rangle \rightarrow \langle \alpha\text{-list-expr} \rangle + \langle \alpha\text{-list-expr} \rangle$
lists can be combined by concatenation

And here are some things we can do with them:

7. $\langle \alpha\text{-var} \rangle \rightarrow \langle \alpha\text{-list-var} \rangle [\langle \text{num-expr} \rangle]$
list elements can be indexed by number
8. $\langle \text{num-expr} \rangle \rightarrow \text{len} (\langle \alpha\text{-list-expr} \rangle)$
a number expression can be the length of a list $\langle \alpha\text{-list-expr} \rangle$

9. $\langle \text{bool-expr} \rangle \rightarrow \langle \alpha\text{-expr} \rangle \text{ in } \langle \alpha\text{-list-expr} \rangle$

a boolean can indicate true if $\langle \alpha\text{-expr} \rangle$ is in $\langle \alpha\text{-list-expr} \rangle$, false otherwise

10. $\langle \text{bool-expr} \rangle \rightarrow \langle \alpha\text{-expr} \rangle \text{ not in } \langle \alpha\text{-list-expr} \rangle$

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
```

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. $\langle \text{stmt} \rangle \rightarrow \text{while } \langle \text{bool-expr} \rangle : \boxed{\text{NEWLINE}} \langle \text{suite} \rangle$

repeat $\langle \text{suite} \rangle$ as long as $\langle \text{bool-expr} \rangle$ is true

2. $\langle \text{stmt} \rangle \rightarrow \text{for } \langle \alpha\text{-var} \rangle \text{ in } \langle \alpha\text{-list-expr} \rangle : \boxed{\text{NEWLINE}} \langle \text{suite} \rangle$

do $\langle \text{suite} \rangle$ for each value in $\langle \alpha\text{-list-expr} \rangle$, assigned to $\langle \alpha\text{-var} \rangle$

For example:

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

will print:

```
1  
2  
3  
4
```

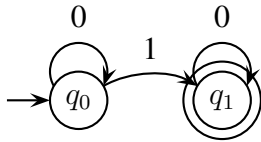
Practice:

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.