CMSI 488 Homework 1

Don Rowe

Due February 21, 2012

1. (a) `/[A–CEGHJ–NPR–TVXY]\d[A–CEGHJ–NPR–TV–Z] \d[A–CEGHJ–NPR–TV–Z]\d/`

   Canadian postal are in the format A0A 0A0, where "A" is any English capital except for D, F, I, O, Q, or U and "0" is any digit. Additionally, the first letter cannot be W or Z. Also, notice the space between the groups of three.

   (b) `/4\d{12}\d{3}?/`

   Visa card numbers begin with a 4 and are followed by either 12 or 15 more digits. The above regular expression assumes that there are always at least the first 13 digits with an optional additional 3 digits.

   (c) `/5[1−5]\d{14}/`

   MasterCard numbers begin with a 5, and the second digit is between 1 and 5. The remaining 14 digits can be any digit.

   (d) `/\d+(_\d+)*#[\dA–F](_[\dA–F]+)*(\.[\dA–F]+(_[\dA–F]+)*)?#(E[\+\−]?\d+(_\d)*)?|\d+(_\d+)*(\.\d(_\d+)*)?(E[\+\−]?\d+(_\d)*)?`

   Numeric literals in Ada95 come in two flavors: based literals and decimal literals.

   A based literal is separated into three sections delimited by # signs. The first section is any number of digits, possibly separated at arbitrary intervals after the first digit by an underscore character; this sequence, represented by `\d+(_\d+)*` in the regular expression, is simply called a numeral. The second section is a based numeral possibly having a fractional part after a dot; based numerals are like numerals, only they are hexadecimal and are represented by `[\dA−F](_[\dA−F]+)*` in the regular expression. The third section is an optional exponent notated with a capital E, an optional plus or minus sign, and a numeral; an exponent is represented in the regular expression by `(E[\+\−]?\d+(_\d)*)`.

   A decimal literal is comprised of a numeral followed by an optional fractional part and an optional exponent.

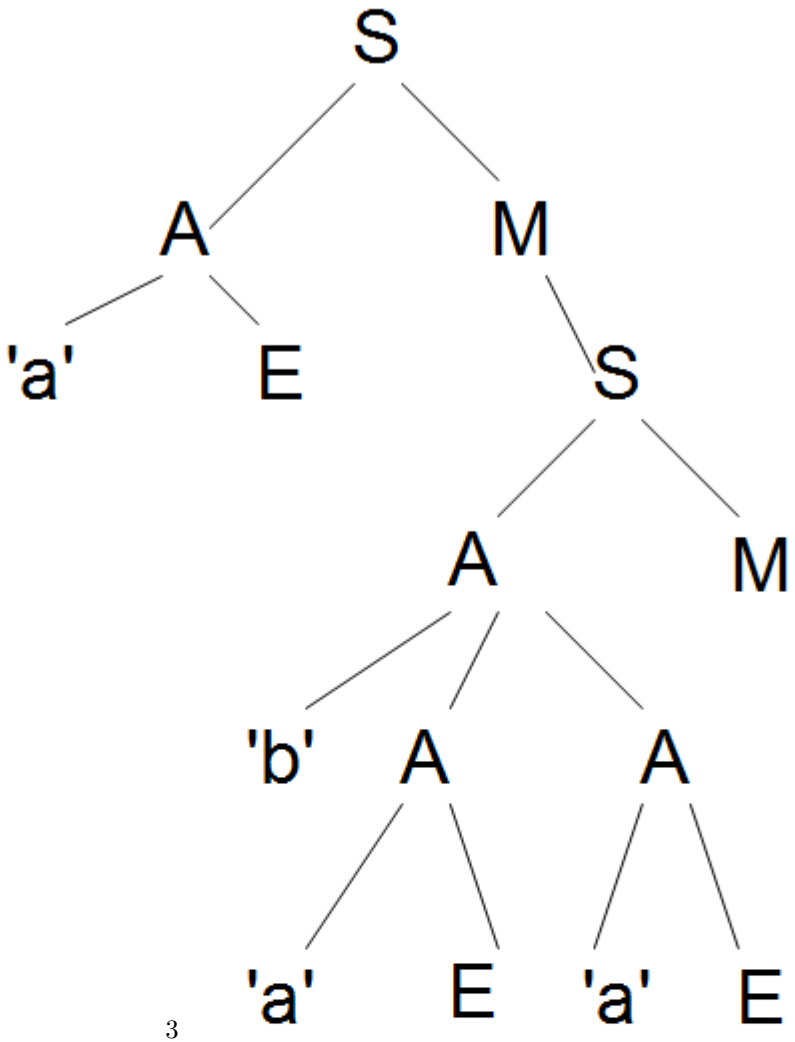
   (e) `/^(c|b(?!ab)|a(?!a))*$/`

   This language includes strings made up of any combination of the characters "a", "b", and "c", provided "a" is never followed immediately by another "a" (represented in the regular expression by `a(?!a)`) and provided "b" is never followed immediately by "ab" (represented by `b(?!ab)`). To ensure that the pattern is applied to the whole string, the beginning ând ending $ are included in the regex; otherwise, legal substrings of an illegal string can be matched. For example, the string accccbabaaccccbcbba does not match as a whole, but the substrings acccc, ab, and accccbcbba (split by the first b and third a) are matched.

2.
```
var program =
{
  'declare': ['x', 'y'],
  'while':
  {
    'condition':
```

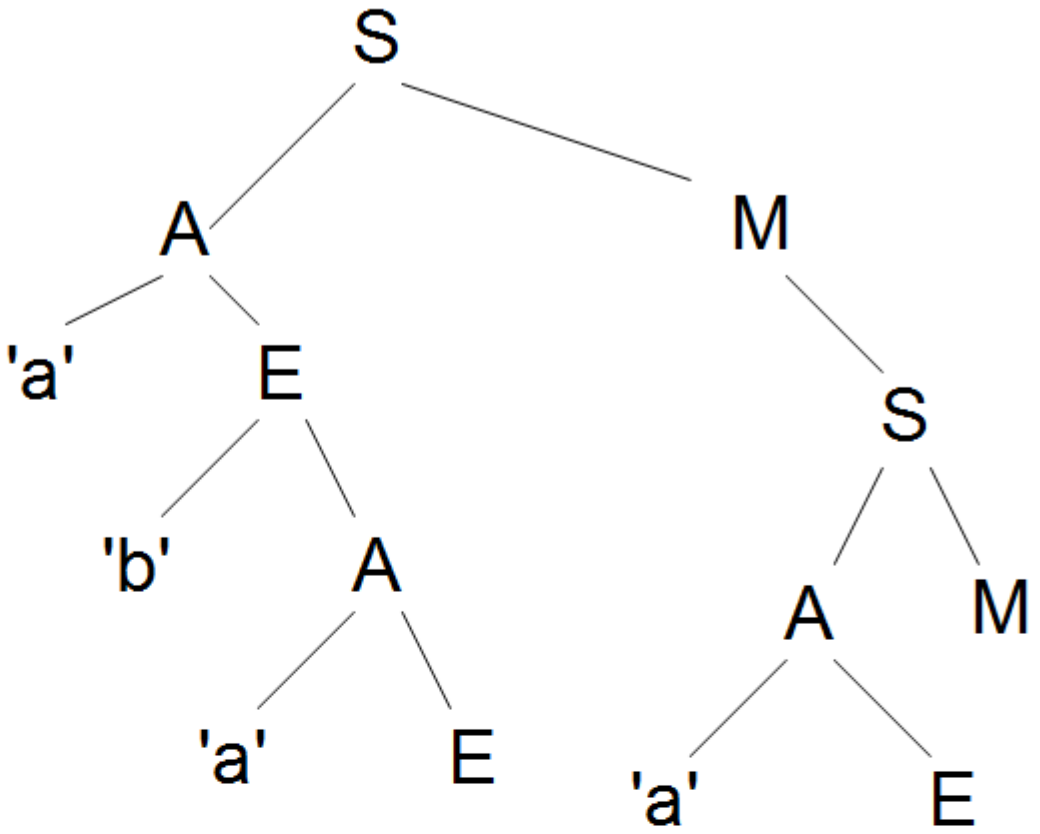
```
    {
      'minus':
      {
        'left': 'y',
        'right': '5'
      }
    },
    'body':
    {
      'declare': 'y',
      'read': ['x', 'y'],
      'assign':
      {
        'left': 'x',
        'right':
        {
          'times':
          {
            'left': '2',
            'right':
            {
              'plus':
              {
                'left': '3',
                'right': 'y'
              }
            }
          }
        }
      }
    },
    'write': '5';
}
```

3. (a) This grammar produces strings made of the characters “a” and “b”. The shortest string in this language is “a”: if a string begins with a “b”, it must be followed by at least “aa”. If a string in this language ends with a “b”, it must be immediately preceded by an “a”.


S
A
M
'a'
E
S
A
M
'b'
A
A
'a'
E
'a'
E

(c) At every choice point of this grammar, the first token of one choice is always ‘a’ and the first token of the second choice is always ‘b’. Thus one only need look forward one token to determine which choice was made. Therefore, this grammar is LL(1).

(d) Grammars are ambiguous if there exists a string for which there is more than one way to parse it. Below is another parse tree for “abaa”. Therefore, this grammar is ambiguous.

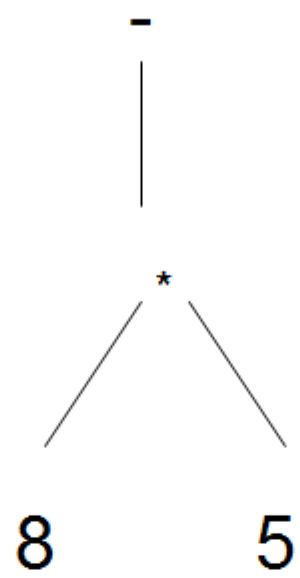

S
A
M
'a'
E
S
'b'
A
A
M
'a'
E
'a'
E

4. (a) At the rule *EXP* lies a choice point with two choices: *ID ":=" EXP* and *TERM TERM_TAIL* . *TERM* expands to *FACTOR FACTOR_TAIL* , and *FACTOR* expands to *"(" EXP ")"* — *ID* . Thus, in *EXP* 's choice point, both choices can begin with an *ID* token, so by looking only one token forward always, one cannot be sure which choice was taken by the generator. Therefore, this grammar is not LL(1).

   (b)
```
EXP          -> ID EXP_TAIL
EXP_TAIL     -> ":=" EXP | TERM_TAIL
TERM_TAIL    -> ("+" TERM TERM_TAIL)?
TERM         -> FACTOR FACTOR_TAIL
FACTOR_TAIL -> ("*" FACTOR FACTOR_TAIL)?
FACTOR       -> "(" EXP ")" | ID
```

5. The unary negation operator was placed with the ADDOPs for symmetry– indeed, unary negation in the form of $-x$ is an implication of $0 - x$. Additionally, since $y - -x$ (that is, subtracting $-x$ from y) cannot occur in Ada, it is consistent to throw out any case of an operator immediately preceding a unary negation operator.

   The expression $-8 * 5$ can be thus represented with this abstract syntax tree:


-
*
8
5

This parsing is similar to parsing with the negation operator in EXP4 in that the answer is still -40 since the negation operator is commutative. The parsing is different, though, in that the negation is applied to the whole product of $8 * 5$ as opposed to applying the negation first to 8.

6. Macro:

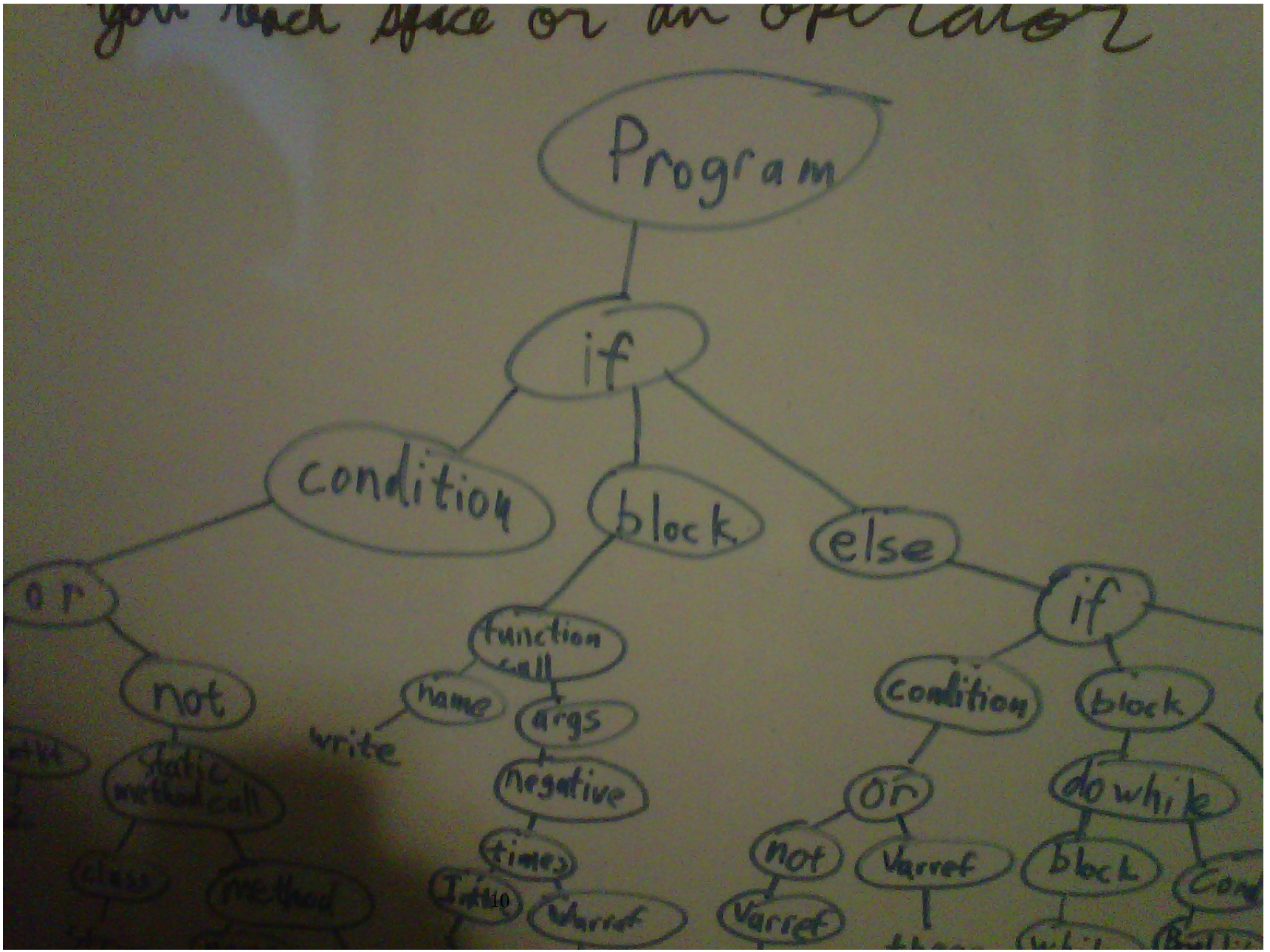
```
PROGRAM      -> (FUN_DECL)+
FUN_DECL     -> SPACE* "fun" SPACE+ ID SPACE* PARAMS SPACE* BLOCK SPACE*
PARAMS       -> "(" SPACE* ID SPACE* ("," SPACE* ID SPACE*)* ")"
BLOCK        -> "{" ( SPACE* EXP SPACE* ";" )+ SPACE* "}"
EXP          -> CONDITIONAL | ARITHMETIC | NUM_LIT | STR_LIT | ID | FUN_CALL
CONDITIONAL  -> EXP SPACE+ "if" SPACE+ EXP SPACE+ "else" SPACE+ EXP
ARITHMETIC   -> TERM ( SPACE* ("+" | "-") SPACE* TERM )*
TERM         -> FACTOR ( SPACE* ("*" | "/") SPACE* FACTOR )*
FACTOR       -> ("-")? SPACE* NEG_OR_POS
NEG_OR_POS   -> EXP SPACE* "!"
FUN_CALL     -> ID SPACE* "(" SPACE* EXP (SPACE* "," SPACE* EXP)* SPACE* ")"
```

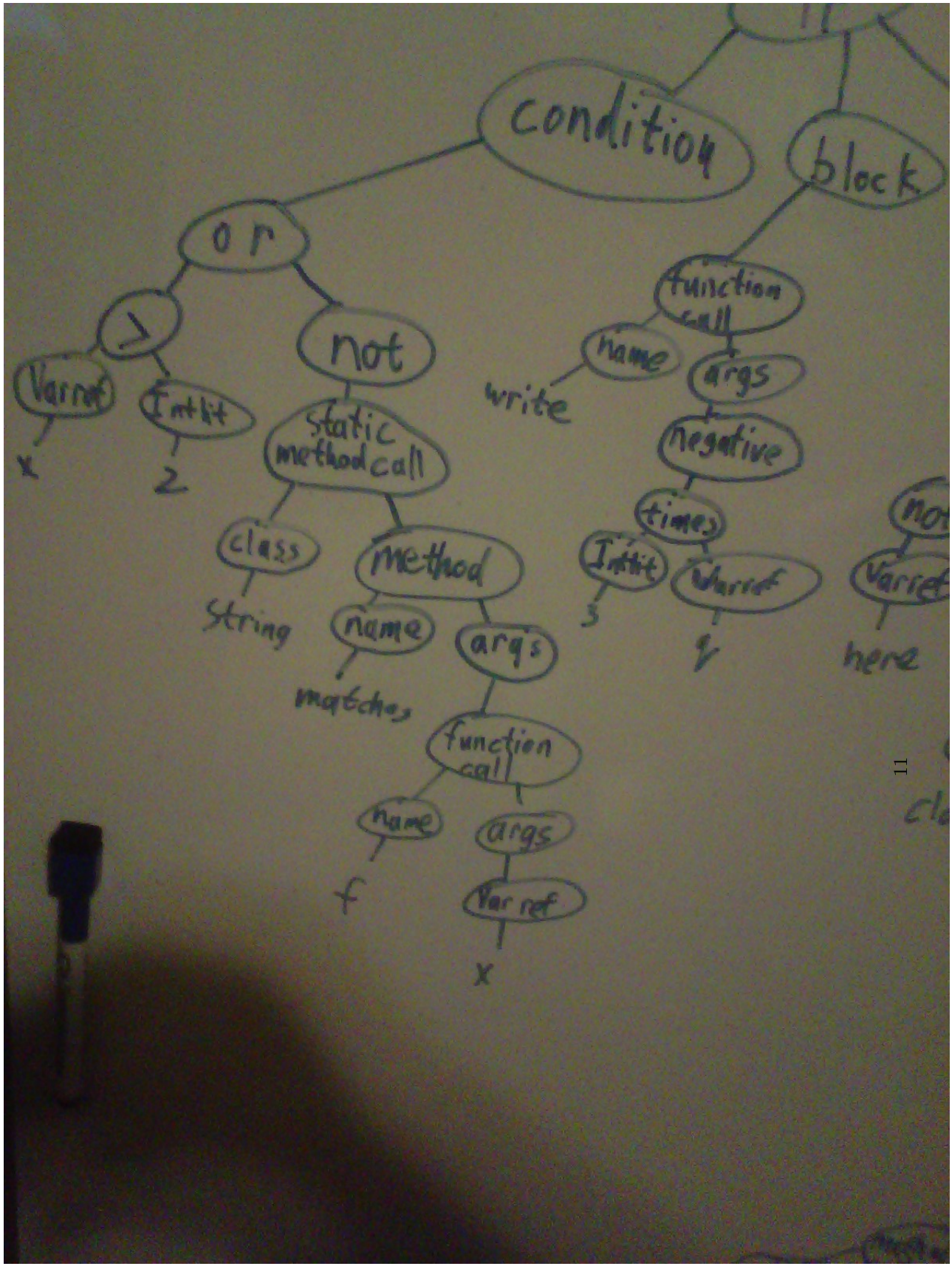
Micro:

```
NUM_LIT      -> DIGIT+ FRACTION? (SPACE* EXPONENT)?
FRACTION     -> "." DIGIT+
EXPONENT     -> "E" SPACE* ("+" | "-")? SPACE* DIGIT+
DIGIT        -> [0-9]
STR_LIT      -> "\""
CHAR         -> DIGIT | ALPHA | PUNC | ESC_SEQ | SPACE
ALPHA        -> [a-zA-Z]
PUNC         -> [-‘~!@#$%\^&*\()_+=|{}[\]';:,./?<>]
ESC_SEQ      -> "\\\"" | "\\'" | "\\r" | "\\n" | "\\\\" | "\\u" HEX{4}
HEX          -> DIGIT | [A-F]
SPACE        -> " " | "\n" | "\t"
ID           -> ( (ALPHA | "$") (ALPHA | DIGIT | [_@$])* ) - RESERVED
RESERVED     -> "fun" | "if" | "else" | "E"
```

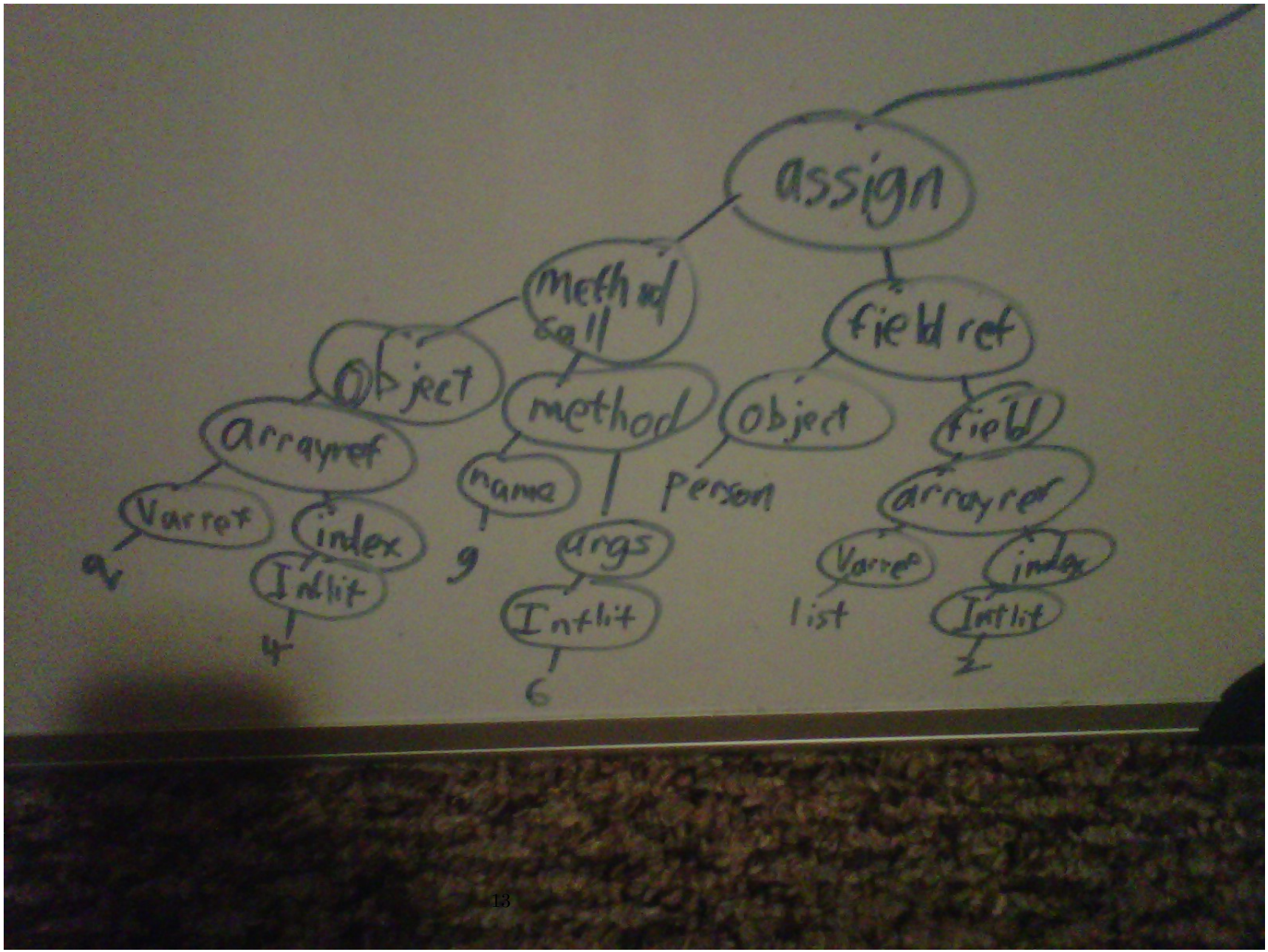

Program
if
condition
block
else
or
>
not
write
name
args
negative
times
Intlit
Varref
2
static method call
class
method
String
name
args
matches
function call
name
args
f
Var ref
x
5
y
here
if
condition
block
else
or
not
Varref
there
do while
block
Cond
while
Boollit
false
cond
block
Varref
close
func call
assign
tryHarder
Var
Exp
x
throw
Exception
Varref
up
assign
Field ref
Object
method
Object
Field
Varref
index
name
args
Person
Intlit
arrayref
Varref
index
Intlit
int
9
4
6
9
6

7.


Program
if
condition
block
else
or
not
function call
name
args
write
negative
times
varref
if
condition
block
or
not
varref
varref
do while
block


if
condition
block
or
<
not
varref
x
intlit
2
static methodcall
class
String
method
name
matches
args
function call
name
f
args
varref
x
function call
name
write
args
negative
times
intlit
3
varref
y
not
varref
here


else
if
condition
block
else
or
do while
throw
not
varref
block
cond
exception
varref
there
while
bool lit
varref
here
false
cond
block
varref
func call
assign
close
tryHarder
var
exp
x
assign
field ref


assign
method call
field ref
object
method
object
field
arrayref
name
person
arrayref
varref
index
args
varref
index
varref
intlit
intlit
intlit
a
9
4
6
list
2

8.
```
#manatee_tokens.py
#Takes Manatee Source as input and outputs the program's token sequence

import fileinput
import re

#The list of tokens in this manatee program
tokens = []

#This pattern matches either a
#string literal, a word char sequence, a newline
#and any of the operators:
token_pattern = r'[\r\n]|(".*(?<!\\)"|\w+\b|[+*/:=;[\]{}<>.,!]|(?<!-)-(?!-))'
token_matcher = re.compile(token_pattern)

#This pattern matches comments:
comment_pattern = r'--[^\r\n]*'
comment_matcher = re.compile(comment_pattern)

#Takes a string of Manatee terminated by at most one newline and returns a list
#with the first token in that string at index 0 and the rest of the
#string at index 1:
def first_token(string):
    global token_matcher
    #Get the leftmost token in this string:
    match = token_matcher.search(string)
    if match:
        token = match.group()

        #index beginning the rest of the string:
        if len(string) == match.end():
            rest_of_string = None
        else:
            rest_of_string = string[match.end():]

        return [token, rest_of_string]
    else:
        return [None, None]


#loop through the lines of the file passed through stdin:
for line in fileinput.input():

    #The current remainder of this line during processing:
    current = comment_matcher.sub("␣", line) #Replace comments with spaces
```

```
    #Keep pulling tokens off the beginning of this string until there are none left:
    while current:
        result = first_token(current)
        #Put the new token in the list:
        if result[0]: #To make sure a string indeed had another token and not just space/comment
            tokens.append(result[0])

        #The remainder of the string after that token:
        current = result[1]

#After the tokens have been sequenced, print them out:
for t in tokens:
    if t in "\r\n":
        print "BR"
    else:
        print t
```