Assignment 3 reading

Sunday, September 24, 2017 8:23 PM

Equal → test two arguments, and returns true if they have the same Expression! Works on atoms & lists

(equal (+ 2 2) 4) → T
(equal '(this is a list) (setf 1 '(this is a list))) → T

Eql → test two argument, returns true if have same Values or Symbols

(eql 4 4.0) → NIL
(eql 4 4) → T

Eq → tests if two arguments share the same bunch of memory; therefore the same symbols

4 4 (Eq 4 4) NIL 4 (Eq 4 4) → T

= → test if two numbers represent the same value

Member → test if the first argument is a top-level element in the second list argument. Return the rest of list w/ argument

(setf sentence '(tell me more about your mother please))
(member 'mother sentence) → mother please

SUMMARY

This chapter introduced the most versatile data type in Lisp: lists. Lists have both a printed and an internal representation. They may contain numbers, symbols, or other lists as elements.

We can take lists apart using CAR and CDR (''first'' and ''rest'') and put them together with CONS or LIST. The LENGTH function counts the number of elements in a list, which is the same as its number of top-level cons cells.

The important points about CAR and CDR are:

- CAR and CDR accept only lists as input.
- FIRST and REST are the same as CAR and CDR.
- SECOND and THIRD are the same as CADR and CADDR.
- Common Lisp provides built-in C...R functions for all combinations of CAR and CDR up to and including four As and Ds.

The symbol NIL has several interesting properties:

- NIL is a symbol. It is the only way to say ''no'' or ''false'' in Lisp.
- NIL is a list. It is the empty list; its LENGTH is zero.
- NIL is the only Lisp object that is both a symbol and a list.
- NIL marks the end of a cons cell chain. When lists are printed in parenthesis notation, the NILs at the end of chains are omitted by convention.
- NIL and () are interchangeable notations for the same object.
- The CAR and CDR of NIL are defined to be NIL.

: → any symbol beginning with a colon is said to be a Keyword. Keywords evaluate to themselves.
→ any argument that follows a keyword is a Keyword Argument. Its purpose is to modify a functions basic behavior suggested by the argument.

#' → produces procedure objects from procedure names → Old form (Function Equal) ≡ #'Equal
defun → stores procedure descriptions as procedure objects under procedure names.
#'<expression> from (Function <expression>)

FUNCTIONS COVERED IN THIS CHAPTER

List functions: FIRST, SECOND, THIRD, FOURTH, REST, CAR, CDR, CONS, LIST, LENGTH.

Compositions of CAR and CDR: CADR, CADDR, and so on.

Predicates: LISTP, CONSP, ATOM, NULL.

:Test indicates that the next argument specifies the test that function is to use

procedure name → Equal, defun, +

procedure object → the actual computer code that actually perform the work.

Beginning Lispers often have trouble distinguishing among CONS, LIST, and APPEND, since all three functions are used to build list structures. Here is a brief review of what each function does and when it should be used:

- CONS creates one new cons cell. It is often used to add an element to the front of a list.
- LIST makes new lists by accepting an arbitrary number of inputs and building a chain of cons cells ending in NIL. The car of each cell points to the corresponding input.
- APPEND appends lists together by copying its first input and making the cdr of the last cell of the copy point to the second input. It is an error for the first input to APPEND to be a non-list.

atom → test a single argument, return T if it is an atom

numberp → test a single argument, return T if it is a numeric atom

symbolp → test a single argument, return T if it a non numeric atom

listp → test a single argument, return T if it is a list

Predicates more...

If →
```
(if <test>
    <then form>
  <else form>
)
```
example:
```
(set f day-or-date 'monday)
(if (symbolp day-or-date)
    'day
  'date
)
→ Day
```
← if <test>
← then return day
← else return date

When →
```
(when <test>
    <then form>
)
```
≡
```
(if <test>
    <then form>
  NIL
)
```
example.
```
(setf high 98 temp 102)
(when (> temp high)
    (setf high temp)
    'new-record)
```
← side effects
← supplies the value of a "True" test case

```
> New Record
> high
→ 102
> 103
→ New Record
```

Unless →
```
(unless <test>
  <else form>
)
```
≡
```
(if <test>
    NIL
  <else form>
)
```

```
(Cond (<test 1> <consequence 1> .... <>)
      (<test 2> <consequence 2> .... <>)
      :
      (<test m> <consequence m> .... <>))
```

Defun w/ Conditionals

```
(defun express-probability (p)
  (cond ((> p .75) 'very likely)
        ((> p .50) 'likely)
        ((> p .25) 'unlikely)
        (t 'very unlikely)))
```

Tuesday, October 24, 2017 8:24 PM

HomeWork 4

```
(defun my-sum (L)
  (let ((x (sum (cdr L))))
    (+ (car L) x)))
```

Returns the sum over a list
X = (sum (rest L))
(+ (First L) (rest L))
//the sum of the rest of the list

X = sum of rest of the list
To return the actual answer we
Add the first E to X

```
(defun my-neg-nums (L)
  (let ((x (neg-sums (cdr L))))
    (if (minusp (car L))
        (cons (car L) x)
        x)))

(defun my-inc-list-2 (L N)
  (let ((x (inc-list-2 (cdr L) N)))
    (cons (car L) x)))

(defun my-insert (N L)
  (let ((x (insert N (cdr L))))
    (if (<= (car L) N)
        (append L (list N))
        (cons (car L) x))))

(defun my-sort (L)
  (let ((x (sort (cdr L))))
    (insert (car L) x)))

(defun my-split-list (L)
  (let ((x (split-list (cdr L))))
    (list (append (list (car L)) (second x))
          (first x))))

(defun my-partition (L P)
  (let ((x (partition (cdr L) P)))
    (if (<= (car L) P)
        (list (append (first x) (list (car L)))
              (second x))
        (list (first x)
              (append (second x) (list (car L)))))))
```

Main Problems

```
(defun Sum (L)
  (if (endp L)
      0
      (+ (car L) (sum (cdr L)))))
```

if the list is empty () = 0
(+ (first L) (sum (rest L)))
if L = (3) then 3 + () => 4
= 0

```
(defun neg-nums (L)
  (cond ((endp L) ())
        ((< (car L) 0) (cons (car L) (neg-nums (cdr L))))
        (else (neg-nums (cdr L)))))
```

if L is empty return ()
if first L < 0 then make list of (first L) & (neg-nums (rest L))

```
(defun inc-list-2 (L N)
  (if (endp L)
      ()
      (cons (+ N (car L)) (inc-list-2 (cdr L) N))))
```

if L is empty ()
make a list of ((first L) + N) (inc-list-2 (rest L) N)

```
(defun insert (N L)
  (cond ((endp L) (list N))
        ((<= N (car L)) (cons N L))
        (else (cons (car L) (insert N (cdr L))))))
```

if inserting N on () then (list N)
if (N <= (first L)) make a list of (N and L)
otherwise make a list of ((first L) and (insert N (rest L)))

```
(defun isort (L)
  (if (endp L)
      ()
      (insert (car L) (isort (cdr L)))))
```

if L is empty ()
otherwise (insert (first L)) into (isort (rest L))

The formation of Split-lists

(list (cons (first L) (second x)) (first x)) alter the second list

or

(list (second x) (cons (first L) (first x))) alter the first list

```
(defun split-list (L)
  (cond ((endp L) '(() ()))
        (else
          (let ((x (split-list (cdr L))))
            (list (cons (car L) (second x))
                  (car x))))))
```

if L is empty (() ()) //make the split-lists

```
(defun partition (L P)
  (if (endp L)
      '(() ())
      (let ((x (partition (cdr L) P)))
        (if (< (car L) P)
            (list (cons (car L) (car x))
                  (second x))
            (list (car x)
                  (cons (car L) (second x)))))))
```

if list is empty (() ())
x = (partition (rest L))

if ((first L) < P) (list (cons (first L) (first x)) (second x))
else ((first L) ≥ P) (list (first x) (cons (first L) (second x)))

```
(defun split-nums (n)
  (if (zerop n)
      '(0 ())
      (let ((x (split-nums (- n 1))))
        (if (evenp n)
            (list (cons n (car x))
                  (second x))
            (list (car x)
                  (cons n (second x)))))))

(defun set-union (s1 s2)
  (cond ((endp s1) s2)
        ((member (car s1) s2) (set-union (cdr s1) s2))
        (else (cons (car s1) (set-union (cdr s1) s2)))))

(defun Set-remove (x s)
  (cond ((endp s) ())
        ((equal x (car s)) (cdr s))
        (else (cons (car s) (set-remove x (cdr s))))))

(defun Singletons (e)
  (cond ((endp e) ())
        ((member (car e) (cdr e))
         (set-remove (car e) (singletons (cdr e))))
        (else (cons (car e) (singletons (cdr e))))))
```

jjhjj

Homework3

```
(defun min-2 (a b)
  (cond ((not (numberp a)) 'Error)
        ((not (numberp b)) 'Error)
        ((<= a b) a)
        (else b)))

(defun safe-avg (x y)
  (if (and (numberp x) (numberp y))
      (/ (+ x y) 2)
      'Nan))

(defun add-gt-million (x)
  (and (integerp x) (> x 1000000) (add1 x)))

(defun multiple-member (x L)
  (if (and (atom 'x) (listp L))
      (member 'x (member x' L))))

(defun month-> integer (m)
  (cond ((equal m 'January) 1)
        ((equal m 'February) 2)
        ((equal m 'March) 3)
        ((equal m 'April) 4)
        ((equal m 'May) 5)
        ((equal m 'June) 6)
        ((equal m 'July) 7)
        ((equal m 'August) 8)
        ((equal m 'September) 9)
        ((equal m 'October) 10)
        ((equal m 'November) 11)
        ((equal m 'December) 12)
        (else 'error)))

(defun score->grade (x)
  (cond ((not (numberp x)) NIL)
        ((>= x 90) 'A)
        ((>= x 87) 'A-)
        ((>= x 83) 'B+)
        ((>= x 80) 'B)
        ((>= x 77) 'B-)
        ((>= x 73) 'C+)
        ((>= x 70) 'C)
        ((>= x 60) 'D)
        (else 'F)))

(defun gt (x y)
  (and (numberp x) (numberp y) (> x y)))

(defun same-sign (x y)
  (and (numberp x)
       (numberp y)
       (or (and (zerop x)
                (zerop y))
           (and (< x 0) (< y 0))
           (and (> x 0) (> y 0)))))

(defun safe-div (x y)
  (and (number p x) (numberp y) (not (zerop y))
       (or (and (zerop x) (zerop y))
           (/ x y))))
```

```
#|kawa:2|# (pretty-print index$)
(define (index n l)
  (cond ((null? l) 'ERROR)
        ((= n 1) (car l))
        (else (index (- n 1) (cdr l)))))
41
#|kawa:3|# (pretty-print min-first$)
(define (min-first l)
  (if (null? (cdr l))
      l
      (let ((min-first-cdr (min-first (cdr l))))
        (if (<= (car l) (car min-first-cdr))
            l
            (cons (car min-first-cdr)
                  (cons (car l) (cdr min-first-cdr)))))))
54
#|kawa:4|# (pretty-print ssort$)
(define (ssort l)
  (cond ((null? l) ())
        (else
          (let ((l1 (min-first l)))
            (cons (car l1) (ssort (cdr l1)))))))
48
#|kawa:5|# (pretty-print qsort$)
(define (qsort l)
  (cond ((null? l) ())
        (else
          (let ((pl (partition l (car l))))
            (cond ((null? (car pl)) (cons (car l) (qsort (cdr l))))
                  (else
                    (append (qsort (car pl)) (qsort (cadr pl)))))))))
68
#|kawa:6|# (pretty-print merge-lists$)
(define (merge-lists l1 l2)
  (cond ((null? l1) l2)
        ((null? l2) l1)
        ((< (car l1) (car l2))
         (cons (car l1) (merge-lists (cdr l1) l2)))
        (else (cons (car l2) (merge-lists l1 (cdr l2))))))
59
#|kawa:7|# (pretty-print split-list$)
(define (split-list l)
  (cond ((null? l) '(() ()))
        (else
          (let ((split-cdr (split-list (cdr l))))
            (list (cons (car l) (cadr split-cdr))
                  (car split-cdr))))))
38
#|kawa:8|# (pretty-print remove-adj-dupl$)
(define (remove-adj-dupl l)
  (cond ((or (null? l) (null? (cdr l))) l)
        ((equal? (car l) (cadr l))
         (cons (car l) (cdr (remove-adj-dupl (cdr l)))))
        (else (cons (car l) (remove-adj-dupl (cdr l))))))
58
#|kawa:9|# (pretty-print unrepeated-elts$)
(define (unrepeated-elts l)
  (cond ((null? l) ())
        ((or (null? (cdr l))
             (not (equal? (car l) (cadr l))))
         (cons (car l) (unrepeated-elts (cdr l))))
        ((or (null? (cddr l))
             (not (equal? (car l) (caddr l))))
         (unrepeated-elts (cddr l)))
        (else (unrepeated-elts (cdr l)))))
43
#|kawa:10|# (pretty-print repeated-elts$)
(define (repeated-elts l)
  (cond ((null? l) ())
        ((or (null? (cdr l))
             (not (equal? (car l) (cadr l))))
         (repeated-elts (cdr l)))
        ((or (null? (cddr l))
             (not (equal? (car l) (caddr l))))
         (cons (car l) (repeated-elts (cddr l))))
        (else (repeated-elts (cdr l)))))
41
#|kawa:11|# (pretty-print count-repetitions$)
(define (count-repetitions l)
  (cond ((null? l) ())
        ((null? (cdr l)) (list (cons 1 l)))
        (else
          (let ((count-rep-cdr (count-repetitions (cdr l))))
            (if (not (equal? (car l) (cadr l)))
                (cons (list 1 (car l)) count-rep-cdr)
                (cons (list (+ 1 (caar count-rep-cdr)) (car l))
                      (cdr count-rep-cdr)))))))
45
#|kawa:12|# (pretty-print subset$)
(define (subset f l)
  (cond ((null? l) ())
        ((f (car l)) (cons (car l) (subset f (cdr l))))
        (else (subset f (cdr l)))))
36
#|kawa:13|# (pretty-print our-some$)
(define (our-some f l)
  (cond ((null? l) #f)
        ((f (car l)) l)
        (else (our-some f (cdr l)))))
38
#|kawa:14|# (pretty-print our-every$)
(define (our-every f l)
  (cond ((null? l) #t)
        (else (and (f (car l)) (our-every f (cdr l))))))
57
#|kawa:15|# (pretty-print qsort1$)
(define (qsort1 p l)
  (if (null? l)
      ()
      (let ((pl (partition1 (cdr l) (car l) p)))
        (append
          (qsort1 p (car pl))
          (cons (car l) (qsort1 p (cadr pl)))))))
48
#|kawa:16|# (pretty-print foo$)
(define (foo f l)
  (if (null? l)
      ()
      (cons (cons (f (car l)) (cdr l))
            (map (lambda (lst) (cons (car l) lst))
                 (foo f (cdr l))))))
35
#|kawa:17|# (pretty-print tr-mul$)
(define (tr-mul l res)
  (if (null? l)
      res
      (tr-mul (cdr l) (* (car l) res))))
39
#|kawa:18|# (pretty-print tr-fac$)
(define (tr-fac n res)
  (if (zero? n) res (tr-fac (- n 1) (* n res))))
49
#|kawa:19|# (pretty-print slow-primep$)
(define (slow-primep n)
  (= (modulo (tr-fac (- n 1) 1) n) (- n 1)))
45
#|kawa:20|# (pretty-print transpose1$)
(define (transpose1 m)
  (if (null? (cdr m))
      (map list (car m))
      (map cons (car m) (transpose1 (cdr m)))))
46
#|kawa:21|# (pretty-print transpose2$)
(define (transpose2 m)
  (if (null? (cdar m))
      (list (map car m))
      (cons (map car m) (transpose2 (map cdr m)))))
50
#|kawa:22|# (pretty-print transpose3$)
(define (transpose3 m) (apply map list m))
43
```

Venus

Saturday, September 2, 2017 2:55 PM

Venus.cs.qc.edu
Swda7958
23347958

Login to euclid through Venus
Ssh swint_darc316@euclid
DSWINTONusmc

Context free grammar

F ::= blank
| tab
| newline

→

T ::= T-T
| F

✓

<F> ::= blank | tab | newline
<T> ::= <F> | <T><F>

<Letters> ::= a|...|z |A|...|Z
<Digits> ::= 1|...|9

<sequence> ::= <sequence><sequence>
| <sequence><letters>
| <sequence><digits>

<main> ::= <letters> | <letters><sequence>

8/28/17

Monday, August 28, 2017 3:21 PM

82817

addition (+ 3 4)
define function (defun myfunc (n)
(+ n 1)
then CLISP will say MYFUNC
(myfunc 17)
18

```
int f (int n)
{
  return n+1;
}
```

Java	Lisp
f(x,y)	(f x y)
myfunc (17)	(myfunc 17)

variables in LISP don't have a type

:q at Break [#] > b/c here you are talking to debugger

give a value to variable
(setf x 9)
9
(+ x)
10
(myfunc x)
10
(+ x (* x 2))
27

functional programming

- Programs in C# & JAVA are usually written in an imperative style. Imperative programs are thought of as specifying a sequence of actions that tell the machine to perform. These actions frequently change the values stored in variables and components of data structures.
- Functional programming is different. A functional program does not specify a sequence of actions, and never changes the values that are stored in variables and data structure components. Variables & data structures of a functional program are immutable. Once a variable is set, given a value, it will have that value for as long as it (the var) exists. Once a data structure has been created, its contents will never change.
- A purely function program consists of a set of functions each of which return a value without doing anything else – These are functions that have no side effects.
  In particular,
  - never change values of variables, and never update values stored in data structures components
  - never do I/O
  - never throw exceptions

To write a functional program
- A functional program can be used by writing any expression that calls one or more of its functions. And/or also calls library functions (+,-,*,/ .....)
- The value of the expression is the result.
- The body of a function in functional program usually consists of just one expression, whos value will be returned when the function is called

In Java, the body of such a function can be written as follows
```
{
  return (an expression which may be quite big)
}
```

Ex: Factorial
we will write a function such that when passed an non-negative int argument, the function returns the factorial of its argument
5! = 5×4×3×2×1 = 120
1! = 1
0! = 1

n! = n * (n-1)!

Java
```
long factorial (int n)
{
  return n==0 ? 1: n * factorial (n-1);     ← conditional expression
```

Lisp
```
(defun factorial (n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

Java
value of c ? e1 : e2 = { value of e1 if c => true; value of e2 if c => false }

ex: n = 9;
n > 8 ? n * 2 : n - 4
value is 18
n < 8 ? n * 2 : n - 4
value is 5

Lisp
(if c e1 e2)
is analogous to
c ? e1 : e2 in C# & JAVA

Recursion & conditional expressions are often used in functional programs
But iteration is not used.
(Note: Lisp does not force you to use functional style, the teacher does
Many lisp programs use do use iteration (ch. 7)

Lisp Assignment 1

1. (* 30 (-7 2))

2. a) (+ (*3 3) (*4 4))
   b) (+ (*3 17) (*4 19))
   c) (- (+ (*12 12 12) (*1 1 1)) (+ (*9 9 9) (*10 10 10)))

3. a) (/ 30 (- 7 2))
   b) (/ 30.0 (- 7.0 3.0))

4. (/(/ (+ 83 83 85 91 97) 5) 1.0)

6.
a) (/ ((+(*-1 (-11)) (sqrt (- (* (-11)(-11))((-4)(1)(-1302))) (* 2 1))
b) (/ ((- (*(-1)(-11)) (sqrt (- (* (-11)(-11)) ((-4)(1)(-1302)))) (* 2 1))

Lisp Functions

first] These two functions → (first '(fast computers are nice)) = Fast
rest] work on set. and will return Error on other arguments (rest '( )) = computer are nice
SetF — assigns values to symbols → (setf ab-list '(A B)) => ab-list = (A B)
— //Lisp assigns values to symbols as it encounters them. (setf meals '(breakfast lunch dinner))
:

Single quote marks → (first (rest '(a b c))) → signal the end of expression evaluation.
Cons → (cons <new first elemen> <list>) → Ex: (cons 'a '(b c)) = (a b c)
Append → (append <list> <list>) → Ex: (append '(a b) (x y)) = (a b x y)

List construct → creates a list out of list argument. The list function does not alter the form of its arguments

Ex: (list (a) (b) (c)) = ((a)(b)(c)) vs (append (a) (b) (c)) = (a b c)
(list '(a) (b) (c)) =

Push → Push element to First in list → returns new list
Pop → Pop element in First position → return popped elem
nthcdr) → trims first n E,
butlast) shorten lists → trims last n E,
last) → trims list to the last E,
length
reverse

a-list
(assoc <key> <association list>)

round → round value to even int

car) same as first
cdr) rest

(cadr '(a b c))
≡ (car (cdr '(a b c)))   ) = (Second (a b c)) = b   cadr
≡ (first (rest '(a b c)))
(third (a b c)) = c   ) or   caddr
:
(tenth (a b ...)) = J   cad...dr

Variable := is a symbol though of a name for a place to store a value
parameters: of a procedure are variable that are bound and that have values assigned to them when that procedure is entered
body := of a procedure consists of the forms that are evaluated when a procedure is used

defun

```
(defun <procedure name> (<parameters>)
      <body>
)
```

defun does not evaluate its arguments, instead only looking at them

9/6/17

Thursday, September 7, 2017 10:00 AM

you will receive email from me tonight or tommorow, it will be sent to your euclid account. H.W. 15,- 20 exercises ~~on~~ pg 36

Functional Programming :

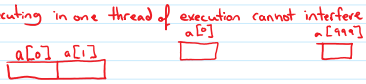
Advantages of functional programming: ① Since variables and data structures are immutable. The code is easier to understand, and reason about, and is quite likely to have fewer bugs.
② Code is easir to test: You never have to worry that a variable or data structure will be changed in inappropriate ways.

In fact. Func prog typically consists mainly of functions that do not use non-local variables and data structures.

A function that has no side effects and does not use non-local variables is called a pure function. The value returned by a call to a pure function depends only on its argument values. So a pure function can be tested just by calling it with different arguments and verifying that the returned values are correct.

③ Code can be automatically parallelized. Different subexpressions can be evaluated in parallel

because code executing in one thread of execution cannot interfere with code in another thread



Lisp is a good language to use for functional programming. Lisp programs often make use of features of features in Lisp that are not used in functional programming — Such as assignment iteration, I/O functionality. However you should not make use of these features in this course.

Lisp was developed by McCarthy and coworkers at M.I.T in the late 1950's. It's the 2nd oldest high level programming language. 1st Fortran
For about 25 years until early 1990's Lisp was the dominant prog lang in A.I.

S-expressions

Expressions in Lisp are called S-expressions. There are two kinds. 1. Actions
2. Lists

The empty list ( ) is both an atom & a list

Atoms include : a) Numbers i.e. 1, 2, 1/3, 3.5
b) Symbols i.e.
c) other kinds of atoms we won't see/use in this course

Numbers (numeric atoms) There are 4 kinds:
1) Integers
2) Ratios // 5/11 or 8/7
Lisp always prints ratios in lowest terms
An integer is equivalent to a ratio w/ denominator 1
3) Floating point numbers
short float
single float = the default 32 bits
double float
long float
4) Complex numbers
#C(3 7) represents 3+7i

Symbols (symbolic atoms)

A variable is a symbol thought of as a name for a place to store a value

Ex: +7= POG x23
symbols are used

- as variable names (including parameter names)
- as function names
- as names of special forms and macros ⌒ user defined function
- as data

Evaluation of Atoms

Evaluation is defined as := computation for the value of

Evaluation of a Symbol:
returns whatever value the symbol may have as a variable

Evaluation of any other kind of atom returns that same atom

Lisp Assignment 2

1 Can an atom be a variable?
Yes, when a symbolic atom is given a value.
Can a number be a variable ?
No, you cannot assign a value to numeric symbols
Can a list be a variable?

Can an atom have no value?
Yes, NIL
Can an atom have more than 1 value at the same time

9/11/17

Monday, September 11, 2017 3:13 PM

Audio Recording...

Last Time:
Lisp expressions are called S-expressions (S= symbolic)

Nil ≡ () empty set

S-expressions are used:
① As Lisp code (i.e. expressions that are to be evaluated by Lisp) Ex: (sqrt (+ (factorial 3) (* 1 1))) = $\sqrt{8}$
② As Lisp data ((name (john smith)) (Job (2005 7 14)))

When an S-expression can be successfully evaluated, the resulting value is always a S-expression : Atoms (ex. Dog, X23, Define, factorial) Symbols : are used as names of variables (not parameters, constants)
Symbols
other kinds (e.x. strings)
numbers
int rats float complex

In (defun f (n)
(* n 2)
)

symbol	defun	is a special form name
"	F	is a function name
"	*	is a function name
"	n	is a variable (formal parameter) name

Lowercase letters in symbol names
automatically converted to uppercase

Evaluation of Atoms

Symbols evaluate to whatever value they have as a variable constant. If a symbol has no value, then it's evaluation produces an error.
all other atoms evaluate to themselves

>73.0
73.0

> 8/6
4/3

Special symbols T & NIL are constants that evaluate to themselves >T T is used to represent "true" however any S-expression other than NIL is true

>NIL NIL ≡ () is used to represent "false"

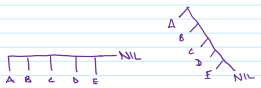
Lists

A proper/true lists has the form: $(E_1\ E_2\ \ldots\ E_k)$ where each E is an S-expression. (Thus E may be an Atom or a List)
where k is the length of the list.
( ) or NIL is the list of length ( )

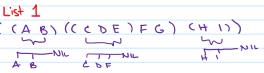
Side note: the construction of these lists are similar to Context-Free Grammars & Regular Expressions

Vine Diagrams / Trees of Lists

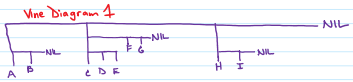
List= (a b c d e) vs. Vine Diagram =



List 1
((A B) (C D E) F G) (H I))



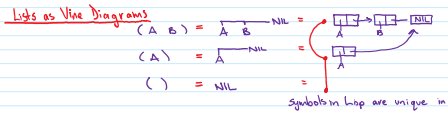
Vine Diagram 1



(A (1 B 2) (A 3))

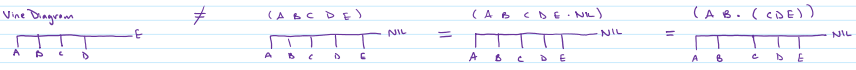
← List
← Vine Diagram

Lists as Vine Diagrams

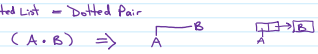


Dotted List

(A B C D . E)
is an ex. of a dotted list



Simplest form of Dotted List = Dotted Pair



A dotted list has the form:
$(E_1\ E_2 \ldots E_k\ .\ a)$
where a is an atom other than NIL

Evaluation of Lists

A list cannot be evaluated unless its first element is:
1) A symbol that is the name of a function (*, + ...)
2) A symbol that is the name of a macro or special form
3) A lambda expression

Which S-expression can be evaluated

Ex:			
	(3 + 4)	no	3 not symbol
	((+ 3 4))	no	(+ 3 4) is not symbol
	(3)	no	3 numeric atom
	(+ 3 4)	yes	

In cases 1-3 how is the list evaluated

Case 1 (F $expr_1$ $expr_2$ $expr_3$)
- check values of arguments
- pass values to function as parameters

if the first element of the list is a function name, then Lisp evaluates each other element of the list, and the calls the function with the values of elements as arguments

Case 2 If the first element of the list is the name of a special form, then the way Lisp evaluates the list depends on the special form/macro

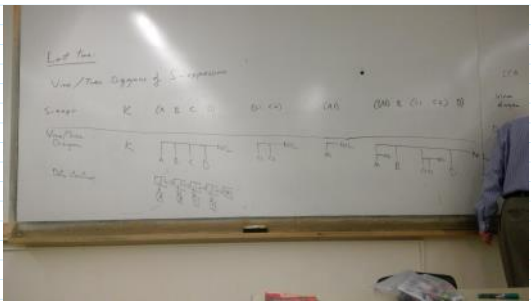
Case 3 > ((lambda (n) (+ n 3)) 17) = 20
n → n+3

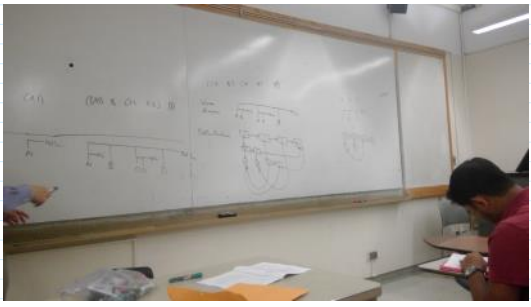
In this case the list is evaluated the same way as Case 1, except the function give by the lambda expression

9/13/17

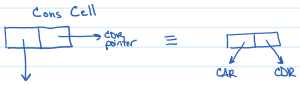
Wednesday, September 13, 2017 3:17 PM

Audio Recording...



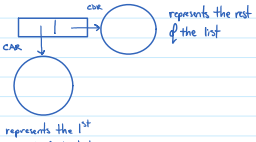


Cons Cell



Any list is represented as a Cons cell connected to 2 other S-expressions

● = Cons cell



The CAR of a list is the first of the list
CDR is the rest of the list (the remaining list after removing CAR)

These are built in functions: CAR $(\mathcal{E}_1\ \mathcal{E}_2\ \mathcal{E}_3)$ => $\mathcal{E}_1$
CDR $(\mathcal{E}_1\ \mathcal{E}_2\ \mathcal{E}_3)$ = $\mathcal{E}_2\ \mathcal{E}_3$

Quoting

(CAR) $(\mathcal{E}_1\ \mathcal{E}_2\ \mathcal{E}_3)$ = $\mathcal{E}_1$

Quote: (') := Special Form
(Quote S-expr)
= S-expr
) ≠ (

Quote does not evaluate S-exp but returns the seen value of S-exp

No Quoting

$(CAR\ (\mathcal{E}_1\ \mathcal{E}_2\ \mathcal{E}_3))$ ≡ Java/C++ $CAR(\mathcal{E}_1, (\mathcal{E}_2\ \mathcal{E}_3))$ → usually an Error unless $\mathcal{E}_1$ is defined as a function name

Exercise:
How to write def/setf to correctly produce
(car (A B C D)) => A

```
(defun a (x y z)
      x )          //return x
(setf b '(a w))    // b = (a w)
(setf c 789 )      //c = 789
(setf d 81 )       // d = 81
```

in Lisp will eval to
(car (A B C D)) → (car (A '(B C D)))
so A was set to a func w/ args BCD and B is the list (A W)
so car receives list (A W) then returns A

(car (A B C D)) => (car (A (B C D)))
(car (B))
(car (a w)) = a

// a(BCD) will return (BCD)
(car(BCD) will return B
B will evaluate to

CAR = Content of the Address part of register
CDR = Contents of the Decrement part of register

C...R functions

(CADR L) = (car (cdr L)) = second element

(CADDR L) = (car (cdr (cdr L))) = third element

(CADDDR L) = (car (cdr (cdr (cdr L)))) = fourth element

(CAAR L) = (car (car (cdr))) = 1st Element of 1st element of L

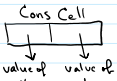
An similarly for other sequences of up to 4 A's & D's between C & R

Use a tree diagram to better understand what is returned

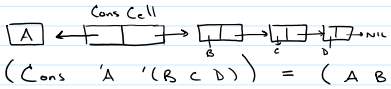
(Nth n L) = n^{th} element of List

Cons, Append, List

Cons : takes two argument
Append: take any # of arguments
List : take any # of arguments
) only one will be correct given the circumstance



(Cons x L) =



(Cons 'A '(B C D)) = (A B C D)

>(Cons '((A) B (C D)) '((P Q) R S (T U)))
(((A) B (C D)) (P Q) R S (T U))

(append L_1 L_2 ..... L_K) => (A B C D ..... X Y Z)
(L_1 = (A B), L_2 = (C D), L_K = (X Y Z))

Append returns a New List obtained by concatenating the Lists passed as arguments to the function. Not changing those lists

(Append '(1 2 3) '(P Q R S)) => (1 2 3 P Q R S)

(List a_1 a_2 .... a_K) => $((a_1)(a_2)\cdots(a_K))$

List of length k whose elements are the Values of the list

(list '(1 2 3) '(P Q R S)) = ((1 2 3)(P Q R S))

9/18/17

Monday, September 18, 2017 1:45 PM

91817

Predicates

In Lisp, a predicate is a function that returns a value of "true" or "NIL". All other S-expressions represents "true"

Ex: (NIL) represents true
(a b c) " "
0 " "

The constant symbol T is the usual representation of "true"

Note: T & NIL evaluate to themselves. So it is unnecessary to Quote T & NIL

⇒ means evaluates to

Ex: > NIL → error
> S → error (unless given a value)

> NIL → NIL
> T → T

Examples of Built-in predicates

(atom x) ⇒ T if the value of x ⇒ atom
⇒ NIL if x ⇒ non empty list

(not x) ⇒ T if x ⇒ NIL
⇒ NIL if x ⇒ anything other NIL ie. any "true" values
used to test if x is false > boolean value

- Internally (not x) ≡ (null x) which reflects the fact that the empty list NIL is how Lisp represents false

(null x) ⇒ T if x ⇒ NIL
= NIL if x ⇒ anything other than NIL
used to test if x is empty list

-However (null x) is used to test whether x is an empty list, whereas (not x) is used to test whether x is false or the negate of a boolean false

(equal x y) ⇒ T if x ⇒ x & y are equal (including in types)
⇒ NIL if x ⇒ x & y are unequal

(eq x y) ⇒ NIL if (equal x y) ⇒ NIL
⇒ T if x,y ⇒ same symbol
→ Analogous to ==
dog1 dog2 ⇒ NIL
→ In many cases the value of (eq x y) depends on which implementation of Common Lisp
Ex: (eq 10,000 10,000) NIL or T depending if Lisp creates a 10,000 or 1
→ As a general rule you should not use (eq — —) unless at least one of the arguments is a symbol

if (equal x y) ⇒ T but the values of x & y is NOT a symbol then (eq x y) may not evaluate to true

Ex: (eq (cons 'A '(B C)) (cons 'A (B C)) ⇒ NIL
(A B C)

Questions

> (atom (+ 3 4))
T

> (atom '(+ 3 4))
NIL

> (null (+ 3 4))
NIL

> (null '(+ 3 4))
NIL

> (equal (+ 1 2) (car '(3 4 5)))
T 3 = 3

> (equal 3.0 3)
NIL

> (equal 1/2 0.5)
NIL

> (equal '((A B) C) (cons '(A B) '(C)))
T

> (eq 3 3.0) ⇒ NIL

> (eq (car '(A B) 'A) ⇒ T

(eql x y) ⇒ NIL if (equal x y) ⇒ NIL
⇒ T if x & y are the same character, number, symbol

Ex: (eql 3.0 3.0) ⇒ T
(eql 1000 1000) ⇒ T

if (equal x y) ⇒ T, but the value of x & y is not a number or character, then (eql x y) returns the same value as (eq x y)

So, as a general rule you should not use eql unless one of the arguments is a symbol, number, or character.

Question (eql 3 3.0) ⇒ NIL

- (= x1 x2 ... xn) another equality predicate but only valid for numbers
(= A A) => error A's are not numbers

- <type> p Predicates : symbolp, numberp, realp, rationalp, floatp
(<type>p x) ⇒ T if x ⇒ a value of type <type>
⇒ NIL if x ⇒ a value that is not of the type <type>

(integerp (+ 4 5.0)) ⇒ NIL ans= 9.0
(listp '(+ 3 4)) ⇒ T ans = (+ 3 4)
(listp (+ 3 4)) ⇒ NIL ans = 7

Warning
(null x) ✓
(atom x) ✓
(null p x) x
(atomp x) x

A CONS is a non empty list
(consp x) => T if x is not NIL but (listp x) ⇒ T
(consp x) ≡ (not (atom x))

Predicates on Numbers → the following predicates will return error is an argument is not a number

(= x y) => T if the values of x & y are Numbers, if one argument is rational while the other is floating point, the float will be converted to rational #

(= 3 3.0) => T b/c (= 3 3)

(= 0.5 1/2) = T

(= 0.2 1/5) ⇒ NIL b/c of rounding error. Whenever you use floating-point representation watch out of rounding error (binary representation of floats)

↳ Test equality of floating point values is dangerous because the result can be affected by rounding error. This reduces the portability of your code.

Unlike equal, eql, eq = can take any number of arguments

(> x1 x2 ... xn) (< 2 3 4) ⇒ T
(<) (< 2 3 1) ⇒ NIL
(>=)
(<=)

Also helps to ensure a certain order of elements

(/= x y) ≡ (not (= x y))

(evenp x) = T if x ⇒ even
NIL if x ⇒ odd
error x ≠ int

(plusp x) ≡ (> x 0) → check if # is positive
(zerop x) (= x 0) → check if # is equal to zero
(minusp x) (< x 0) → check if # is negative

(endp x) ⇒ T if x ⇒ NIL
= NIL if x ⇒ non empty list
error x is anything other than list

Reminder

(if c x1 x2) → c? e1 e2

Ex: (defun kind (e)
(if (eql e 0)
'(integer-0 0)

> (kind (- 3 3)) ⇒ (integer-0 0)
> (kind (+ 3 4)) ⇒ (num -but- not-0 7)

(if c x_1 x_2) → c? e_1 e_2

Ex:
```
(defun kind (e)
  (if (eql e 0)
      '(integer-0 0)
      (if (numberp e)
          (list 'num
                (if (minusp e)
                    'but-not-0
                    '-))
          (if (consp e)
              (list 'non-empty-list e)
              (list 'non-numeric-atom e)))))
```

```
> (kind (- 3 3)) => (integer-0 0)
> (kind (+ 3 4)) => (num -but-not-0 -)
> (kind '(+ 3 4)) => (non-empty-list (+ 3 4))
> (kind 'zero)
```

function name parameter list

```
(defun kind-of-element (e)
  (if (eql e 0)
      '(integer-is-0 0)
      (if (numberp e)
          (list 'num '-)
          (if (consp e)
              (list 'non-empty-list e)
              (list 'non-numeric-atom e)))))
```

9/25/17

Monday, September 25, 2017 3:08 PM 92517 Audio recording started: 3:13 PM Monday, September 25, 2017

Predicates you should now be familiar with include:

(atom x)
(equal x y)
(eq x y)
(eql x y)

(null x) ≡ (eq x NIL)
(not x)
(symbolp x)
(integerp x)
(listp x)
(consp x) → Is x a non-empty list ≡ (not(atom x))

(floatp x)
(realp x)
(rational x)

Predicates on Numbers → every argument must be a number

(= x1 ... xn) → there will be type conversion for the argument
(< x1 ... xn)
(> x1 ... xn)
(<= x1 ... xn)
(>= x1 ... xn)

(plusp x) x > 1
(minusp x) x < 0
(zerop x) x = 0
(evenp int)
(oddp int)

(/= x1 x2) ≡ (not (= x1 x2))

(endp x) = (null x) if x = a list
But if x = an atom (other than NIL) then the evaluation of (endp x) produces an error

So if you "know" x => a list, then it is better to use (endp x) rather than (null x) to test whether x=> an empty list.

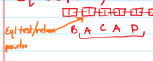
Advantages
- Anyone reading (endp x) will be able to see at once that the value of x must be a list
- If you are wrong, and x=> an atom other than NIL, then calling (endp x) will immediately produce an error whereas (null x) will return NIL

(member x L) is used to test whether x=> a member of list L. Assuming x=> character, number, symbol and L=> a list

Note: (Member x L) use the Eql predicate to compare x's value to elements of L

(member x L) => NIL if symbol not in list
(member x L) => T will return the part of the list that begin w/ first occurrence of x as a list (...)

Eql uses pointers, that why it returns the rest of list



(member 'A '(B C D E)) → NIL
(member 'A '(B A C A D)) → (A C A D)

Ex: (eql (cons 'A '(B C)) '(A B C)) => NIL

(member (cons 'A '(B C)) '(X Y (A B C) Z)) => NIL
↓
(A B C)

Eql works by comparing pointers → Java a == b

: Member does not return T to indicate "true" because it is just as easy for member to return the part of the list that begins with the first occurrence of that element and that sublist will sometimes be useful

You can force Member to use Equal rather than Eql as follows:

(member (cons 'A '(B C)) '(X Y (A B C) Z))
:test #'Equal
=> (A B C) Z

Recall

(if c e1 e2)

Indentation:

```
(if c
    e1
    e2
)
```

(if c e) ≡ (if c e NIL)
(when c e) ≡ (if c e)
(unless c e) ≡ (if (not c) e)

Cond:

(cond (g1 e1)
      (g2 e2)
      (gn en))

≡

(if g1
    e1
    (if g2
        e2
        :
        (if gn
            en) ...)))

guard/test forms — consequent forms

If every guard if false, then the value of cond is NIL

if
else if

```
(defun kind (e)
  (cond ((eql e 0) '(integer-0 0))                          1
        ((numberp e) (list 'numb-but-not-0 e))              2
        ((consp e) (list 'nonempty-list e))                 3
        (t (list 'non-numeric-atom e))))                    4
```

> (kind (- 4 4)) → (integer-0 0)
> (kind (- 4 3)) → (Numb-but-not-0 1)
> (kind (-3 3.0)) → (Numb-but-not-0 00)
> (kind '(-4 4)) → (nonempty-list (-4 4))
> (kind (car '(one two three))) → (non-numeric-atom one)

It is very common for the last guard cond to be the constant T

And & Or

(or e_1 .... e_n) is analogous to (e_1 || .... || e_n) in Java

(and e_1 .... e_n) (e_1 && ..... && e_n)

```
( or (member 'A '(B A P))
     (member 'B '(B A B))
)
→ (A P)
```

However, when the value of (or e_1 .... e_n) is "true", then its value is the value of e_1

when the value of (and e_1 .... e_n) is "true", then its value is the value of e_n

```
( and (member 'A '(B A P))
      (member 'B '(B A P))
)
→ (B A P)
```

10/2/17

10217

Monday, October 2, 2017 3:11 PM

Audio recording started: 3:13 PM Monday, October 2, 2017

Let & Let*

```
(defun h (w x y)
  (let ((x (round (sqrt (x))))
        (y (* x 2))
        (z (+ x y)))
    (/ (+ x y z) w)))
```

round($\sqrt{2}$) → 1

// after the variable enters the let they become shadowed. And original value can't be used.

with Let* the new created variable immediatly shadow.

```
> (h 3 2 6)
```

13/3

```
(defun h* (w x y)
  (let* ((x (round (sqrt (x))))
         (y (* x 2))
         (z (+ x y)))
    (/ (+ x y z) w)))
```

x = 1
y = 2
z = 2+1 = 3

1 2 3 / 3

```
> (h* 3 2 6)
```

2

```
(let ((v1 expr1)
      (v2 expr2)
      (vn exprn))
  body-expr)
```

The Let is evaluated as follows:

1. Evaluate expr1 .... exprn in the environment that exists outside the Let
2. Evaluate the body-expr in an environment in which the var v_i is bound to the $expr_i$ value.
3. The value of the body-expr is returned as the value of Let expr

[$v_1, \ldots, v_n$ / Body-expr]

once outside the Let, the v's are unusable

```
(let ((v1 expr1)
      (v2 expr2)
      (vn exprn))
  body-expr)
```

is roughly equivalent to:

```
(defun my-func (v1,...,vn)
  body-expr)
+
(myfunc expr1 .... exprn)
```

more accurately the Let is:

```
((lambda (v1 v2 ... vn) body-expr) expr1 expr2 .... exprn
```

To define a func without a name use a lambda function

```
((lambda (x y z) (/ (+ x y z) w))
```

↑ no name

```
(round (sqrt (x)))
(* x 2)
(+ y z)
```

$(x, y, z) \mapsto (x+y+z)/w$

Let* is equivalent to nested Lets

```
(let* ((v1 expr1)
       (v2 expr2)
       (vn exprn))
  body-expr)
```

≡

```
(let ((v1 expr1))
  (let ((v2 expr2))
    ...
      (let ((vn exprn))
        body-expr)...))
```

Thus in

```
(let* ((v1 expr1)
       (v2 expr2)
       :
       (vn exprn))
  body-expr)
```

each expression $expr_i$ ($i \geq 2$) is evaluated in an environment that includes bindings of $v_1, \ldots, v_{i-1}$

Two good reasons for using Let & Let*

1. To give meaningful names to certain values (so as to make your code readable

```
(let ((birthdate (cadadr datalist)))
  .... birthdate ....)
```

2. To be able to use the value of a computationally expensive more than once without evaluating more than once.

```
(let ((x (expensive y)))
```

(list expensive y expensive y expensive y) = (list x x x) (/ (+ (x) x))

Recursive Functions

Ex 1 Write a funct list-length such that:

if L ⇒ any list
then (list-length L) ⇒ the # of elements of the list

```
> (list-length '((A B C) (D) (E F) ((G H) I))) ⇒ 4
```

To write a recursive function we first assume the function has already been written.

→ Here we assume List-length has already been written.

Now we can write a non-recursive func mylistlength, such that:

→ If L ⇒ non-empty list
then (mylistlength L) ⇒ the same result as (list-length L)

```
(defun mylistlength (L)
  (let ((x (list-length (cdr L)))
    (+ x 1)))
```

one less off the list

(listlen (B C D E)) = 4

Suppose L = (A B C D E)
Then x ⇒ 4
We want (mylistlength L) ⇒ 5, so x+1 = 5

now we can add a case to this that works on all cases, when a list is empty as well

```
(defun bettermylistlength (L)
  (if (endp L)
      0
      (let ((x (list-length (cdr L))))
        (+ x 1))))
```

Now rename the function to list-length

```
(defun list-length (L)
  (if (endp L)
      0
      (Let ((x (listlength (cdr L))))
        (+ x 1))))
```

→

```
(if (endp L)
    0
    (+ (list-length (cdr L)) 1))
```

10/4/17

Wednesday, October 4, 2017 3:05 PM Audio recording started: 3:11 PM Wednesday, October 4, 2017

```
(defun list-length (L)
   (if (endp L)
       0
       (let ((x (list-length (cdr L))))
         (+ x 1))))
```

```
(defun list-length (L)
   (if (endp L)
       0
       (+ (list-length (cdr L)) 1)))
```

b/c is used just once, there is no need for list so can change code to:

Another example: We'll write a func factorial such that: (factorial n) => n x (n-1) x ... x 2 x 1
(factorial 0) => 1

whenever n is a non-negative int
//be careful of the small cases

```
(defun factorial (n)
   (if (zerop n)  1
       (let ((x (factorial (- n 1))))
         (* x n))))
```

General Pattern:

Let (f e) be a function whose argument e is either a list, of a kind such that the (cdr e) is also a valid list or a non negative integer, then you well may be able to write the func f, in such a way

```
(defun f (e)
   (if ((endp e)        -> list
        (zerop e))      -> number
       ..... -------> expr => (f nil) or (f 0)
       (let ((x (f (cdr e))
             (f (- e 1))))
         (*)))
```

template

(*) expression that compute (f e) from x and possibly e

Note 1) the (*) expr may have more than one case!! → it may be a cond, or an if, expr

2) If you see any case(s) where x does not need to be used (even if x actually is used), then move that/those cases out of the Let, so as to avoid making unnecessary recursive calls (to compute x) in those cases.

3) If there is no cases in which x is used more than once, then eliminate the Let and substitute for each occurence of x

```
(f (cdr e))
(f (- e 1))
```

Write a function Evens such that if L => list of numbers then (evens L) => a list obtained from the list L by cutting out all odd elements

```
> (evens ()) => NIL
> (evens '(2 1 7 4 5 3 6 8)) => (2 4 6 8)
```

```
(defun evens (L)
   (if (endp L)
       NIL
       (let ((x (evens (cdr L))))
         (*))))
```

(*) Expression that computes (evens L) from x and, possibly, L

To write the (*) expr, we'll try examples
→ Suppose L = (2 1 7 4 3 6 8)
→ the x => (4 6 8)
next the (*)-expr => (2 4 6 8)
a good (*)-expr for this problem => (cons (car L) x) → if first elem is even

let's try an example where (car L) is odd
→ Suppose L = (3 4 0 1 6 5 2)
x => (4 6 2)
(*)-expr => (4 6 2)
a good (*)-expr is x

```
(defun evens (L)
   (if (endp L)
       NIL
       (let ((x (evens (cdr L))))
         (if (oddp (car L))
             x
             (cons (car L) x)))))
```

b/c x is only used once we can eliminate the Let

```
(defun evens (L)
   (if (endp L)
       0
       (if (oddp (car L))
           (evens (cdr L))
           (cons (car L) (evens (cdr L))))))
```

Write a Safe-Sum such that: if L => a list of numbers
then (safe-sum L) => the sum of list L

if L => a list whose elements are not all numbers
then (safe-sum L) => error

```
> (safe-sum '(1 7 2 5 4))      => 19
> (safe-sum '(1 7 2 Six 4))    => error
> (safe-sum ())                => 0
```

```
(defun safe-sum (L)
   (if (endp L)
       0
       (let ((x (safe-sum (cdr L))))
         (+ (car L) x))))
```

★ expr that computes (safe-sum L) from x, possibly, L also

To write the ★ expr we'll try examples
Suppose L = (1 7 2 5 4)
then x = (7 2 5 4) = 18
want ★-expr => 1 + (7 2 5 4) = 19
↳ (+ (car L) x)
↳ this is good when x is not "error"
↳ we also see that if (car L) is not a number or if x is not a number "error" is a good ★-expr

```
(defun safe-sum (L)
  (if (endp L)
      0
      (if (not (numberp (car L)))
          'error
          (let ((x (safe-sum (cdr L))))
            (if (eq x 'error)
                'error
                (+ (car L) x))))))
```

Functions of 2 or more arguments

In simple cases (such as this 4 func with 2 args) just one of the arguments needs to change in the recursive calls

Pattern (assuming the 2nd arg will change in the recursive calls)

```
(defun f (e1 e2)
  ((if (zerop e2))
   (if (endp e2)))
    expr that => (f e1 0) or (f e1 nil)
  (let ((x (f e1 {(- e2 1)
                  (cdr e2)})))
    expr that computes (f e1 e2)
    from x, and possibly, e1 for e1
    ))))
```

Ex: Write a function Prepend-2 such that

(prepend-2 L1 L2) returns the same result as (append L2 L1)

```
(defun prepend-2 (L1 L2)
  (if (endp L2)
      L1
      (let ((x (prepend-2 L1 (cdr L2))))
        (cons (car L2) x))))
```

10/11/17

Wednesday, October 11, 2017 1:44 PM

Audio Recording...

Audio recording started: 1:44 PM Wednesday, October 11, 2017

101117

Audio recording started: 2:51 PM Wednesday, October 11, 2017

Simple Recursive Functions

```
( defun f (e)
   ( if { (endp e) }
        { (zerop e) }
     → Some expression for ⇒ (f 0) or (f nil) case

   (let (( x (f { (cdr e)
                { (- e 1)
        expr that computes
          ( f e )
        from X and, possibly, e
   )))
```

→ If there are 2 or more Args often just one of them needs to change in the recursive call, and essentially the same pattern can be used

Single Argument

More Sophisticated Recursion

Sometimes we may want to produce a smaller argument for the recursive call in a different way than by using (cdr e) or (- e 1).
Common Alternatives Include:

(cddr e) Split List could be written this way
(- e 2) Split Num could be written this way

(floor e 2) $= \lfloor \frac{e}{2} \rfloor$ = e >> 1 (right shifted) for int e in Java
= e/2 for ints e ≥ 0 in Java

Note if $e \le 0$, then $\lfloor \frac{e}{2} \rfloor \neq e/2$ in Java b/c $\lfloor -\frac{1}{2} \rfloor = -1$; So if e might be a - int then -1 needs to be dealt w/ as base case if we use (floor e 2) to produce a smaller Arg

- (cdr e1), where e1 is a suitable transformation of e

Pattern:

```
( defun f (e)
   ( if  e is a base case )
     ...
   ( let (( x  ( f ( smaller-version-of  e ))))
```

- If you see cases which X need not be used in the *-expr then move those cases out of the Let avoid making unneeded recursive call
- If there is no case in which X is used more than once then you can eliminate the Let

General Case → expr that computes (f e) from X and, possibly e)))

- Make sure your base case code deals w/ all cases in which the general case could not work

(see BC1 - BC3 checks in assignment 5)

To write the *-expr lets try an example
the list e = (A B C D E F G)
we want our X = ((C E G) (D F))
This way the *-expr = ((A C E G) (B D F))
An expression that can produce the *-expr looks like:

```
( list  ( cons (car e) (car X))  ( cons (cadr e) (cadr X))
( list  ( cons (first e) (first X))  ( cons (second e) (second X))
                          or
```

To determine the base case from the general case, when will this code not work

when e = (A)
then X = (nil nil)
* ⇒ ((A) nil) ; but the above * will produce ((A) (NIL)) = wrong!

So for this case (list e nil) ⇒ ((A) nil) = Right

Since X is not used in this case, this can move out the Let, and be a base case also
when e = () = a base case since (cddr e) = e
↳ In this case (split-list e) should give ⇒ (nil nil). And (list e nil) works again

The predicate (endp (cdr e)) can be used to test whether e = a list of length ≤ 1
(endp (rest e))

```
( defun split-list (l)
   [ base case code ]
   ( let (( x ( split-list (cddr l))))
     expr that computes
       (split-list e)
     from X and/or e
   )))
```

```
( defun split-list (L)
   ( if ( endp ( rest e))
        ( list e nil)
        ( let (( x ( split-list (cddr L))))
             (list (cons (first e) (first x))
                   (cons (second e) (second x)))))
```

↳ create split-nums using as exercise (- e 2)

Example: Write a function power such that: y ⇒ a number & n ⇒ positive integer
then (power y n) ⇒ y^n

The idea is $y^n = (y^{\lfloor \frac{n}{2} \rfloor})^2$ if n is even
$y^n = (y^{\lfloor \frac{n}{2} \rfloor})^2 y$ if n is odd

n=1 is a base case
n=0 is not a valid arg $\lfloor \frac{n}{2} \rfloor = 0$, which will not be a valid arg val for us.

```
( defun power ( y n )
   (if  (= n 1)
        y
        (let (( x (power y (floor n 2))))
          (if (evenp n)
              (* x x)
              (* x x y)))))
```

Depth of recursion = (no. of bits in the binary representation of n) - 1

SSort is H.W # 5

(ssort L) => a list of the numbers in L in ascending order

(SSort '(5 4 4 1 6 6 1 2 5 6 6 2))

→ (1 1 2 2 4 4 5 5 6 6 6 6)

(Min-First '(5 4 4 1 6 6 1 2 5 6 6 2))

→ (1 5 4 4 1 6 6 1 2 5 6 6 2)

e1

Compute from SSort (cdr e1) where e1 = min-first

10/16/17

Friday, October 27, 2017 8:27 PM

last time:

Alternative ways of producing a smaller argument for a recursive call:

- (cddr e) = (rest (rest e))
- (- e 2)
- (floor e 2) = $\lfloor \frac{e}{2} \rfloor$ This can be used for when e is an Exponent being reduced
- (cdr e1) where e1 is a suitably transformed version of e

(ssort L) => a list where elements are those of L, in non-decreasing order

(ssort L) can be computed from (ssort L1) where L1 = (minfirst L)

↳ L has the first occurrence of its least element in the front

L = (4 4 1 6 6 1 8 2 5 6 6 27)

L1 = (1 4 4 6 6 1 8 2 5 6 6 27)

cdr L1

Sometimes it is appropriate to change more than one argument for a recursive call

Example: Index from HW 5

Another Example: Write a function Power such that if y = a number & n = positive int then (power y n) = y^n

Method 1 $y^n = (y^{\lfloor \frac{n}{2} \rfloor})^2$ if n is even & $y^n = (y^{\lfloor \frac{n}{2} \rfloor})^2 y$ if n is odd

Base Case

n=1 => $y^n = y$

```
(let (x (power y (floor n 2))))
  (if (evenp n)
      (* x x)
      (* x x y)))
```

Method 2 $y^n = (y^2)^{\lfloor \frac{n}{2} \rfloor}$ if n is even & $y^n = (y^2)^{\lfloor \frac{n}{2} \rfloor} y$ if n is odd

base case $y^n = y$

```
(defun power (y n)
  (if (= n 1)
      y
      (let ((x (power (* y y) (floor n 2))))
        (if (evenp n)
            x
            (* x y))))))
```

→

```
(defun power (y n)
  (if (= n 1)
      y
      (power (* y y) (/ n 2))
        (* (power (* y y) (floor n 2)) y))))
```

$y^n = y^{n-1} y$ is another recursive strategy, but the recursive depth would be high whereas the depth of the $\lfloor \frac{n}{2} \rfloor$ = n >> 1

Strategy is (number of bits in n) - 1

Sometimes it is appropriate to use different recursive call for different argument values

Ex: Merge-List in Assigment 5

(merge-list L1 L2) => a single list whose elements are those of L1 and L2 in non-decreasing order

L1 = (1 1 2 2 3 3 4 8 10 11)

L2 = (5 6 6 6 9 10 10 10)

(merge-list L1 L2)

=> (1 1 2 2 3 3 4 5 5 6 6 6 8 9 10 10 10 10 11)

(merge-list L1 L2) may be computed from 1st (merge-list (cdr L1) L2))

2nd (merge-list L1 (cdr L2))

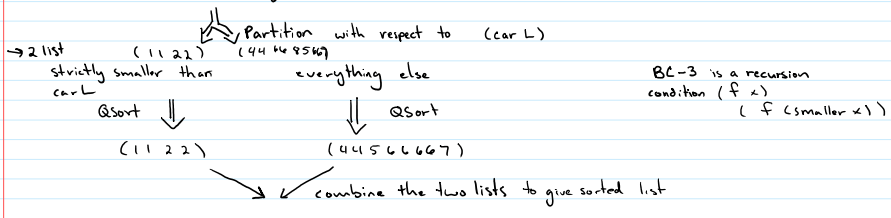
101817

10/18/17

Wednesday, October 18, 2017 3:09 PM Audio recording started: 3:12 PM Wednesday, October 18, 2017

QSort example of double recursion

L = (4 4 1 6 6 1 8 2 5 6 6 2 7)



BC-3 is a recursion condition (f x) (f (smaller x))

Errors
BC-1
BC-2

Warning:

It is possible the first list used by partition will be empty, and the second list hold all the elements (example of a BC-3 case)
// in this case you will continue to call QSort on a list equal to the first list. Because the argument for the recursive call is not smaller than the original list. so we must treat this possibility as a base case of the above recursive strategy.

→ an alternative approach is to use the (partition (cdr L)) with respect to (car L) → This avoids the BC-3 case above.) You would of course have to put the car of L into the result list.

Functions that take functions as arguments

like QSort1 in Assignment 5

Sigma Notation $\sum_{j=2}^{7} f(j) = f(2)+f(3)+f(4)+f(5)+f(6)+f(7)$

How can we write a func Sigma such that: (sigma #'f 2 7) ⇒ $\sum_{j=2}^{7} f(j)$

if f is the name of a function that takes 1 numerical argument

(defun f (n) (* n 3))
(sigma #'f 2 7) = $\sum_{j=2}^{7} f\,3j = 3\times2 + 3\times3 + 3\times4 + 3\times5 + 3\times6 + 3\times7 = 81$

Why the #' ??

In Lisp (as well as Java) the value of a symbol (as a variable) is a different property of that symbol from its function definition.

p may have both a value and a function definition
or a value but no function definition
or no value but have a function definition
or neither

Java	Lisp	
p + 3	(+ p 3)	uses value of p
p(7)	(p ...)	uses function definition of p

(sigma f 2 7) would use the value of f not the function definition

#'f = (function f) => function definition of f → similar to → 'f = (Quote f)

to set the value of a symbol is setf, or (defun f (n) (* n 3)) and running (f 19) now n = 19
defun, sets the function definition of a symbol

(defun f (n) (* n 3))
(setf f 9)

if run
(sigma f 2 7) → you will get an Error because 9 is not a function and so for the 1st parameter of Sigma

Example:

(setf f #'sqrt)
(defun f (n) (* n 3))

so (Sigma f 2 7) ⇒ $\sqrt{2} + \sqrt{3} + \sqrt{4} + \sqrt{5} + \sqrt{6} + \sqrt{7}$

```
(defun f (n) (* n 3))
```

so (Sigma f 2 7) ⇒ $\sqrt{2} + \sqrt{3} + \sqrt{4} + \sqrt{5} + \sqrt{6} + \sqrt{7}$

(Sigma #'f 2 7) ⇒ 3·2 + 3·3 + 3·4 + 3·5 + 3·6 + 3·7

Example:

```
(defun g (x) (+ x 5))
(setf g 3)

>(g g) → 8
     ↓
     3
```

Rule:

→ If a symbol p is the first element of a List, then the function definition of p is used by Lisp when evaluating that list.

→ If it is not the first element, then the value of P will be used

FunCall

(Funcall f $e_1 \dots e_k$)

calls the function given by the value of f, then passes elements $e_1 \cdots e_k$ as arguments to f

Example:

```
(defun g (x y z) (* x y z))
(setf g #'+)

(g 2 3 4) => 2·3·4 = 24
(funcall g 2 3 4) => 2+3+4 = 9
(funcall #'g 2 3 4) => 2·3·4 = 24
```

Note: In Scheme, function definition is just a special case of "value"

How can we write Sigma

```
(defun sigma (f m n)      => f(m) + f(m+1) + f(m+2) ... f(n)
  (if (> m n)
      0
      (let ((x (sigma f (+ m 1) n)))
        (+ (funcall f m) x))))
```

supposed to return

+the base case last

try solve general case first

Mapcar

Built in function

```
(defun f (x) (* x 7))
```

→ (mapcar #'f '(0 1 0 2 3 -1 6)) → returns a list with the function definition applied to every element

⇒ (0 7 0 14 21 -7 42)   (funcall g 0)

→ (mapcar #'car '((A B C) (D E) ((F G) H I) (J)))

⇒ (A D (FG) J)

→ (mapcar #'cdr '((A B C) (DE) ((FG) H I) (J)))

⇒ ((BC) (E) (I J) ())

(funcall g '(A B C))

Our Own Version of mapcar

```
(defun our-mapcar (g L)
  (if (endp L)
      nil
      (let ((x (our-mapcar g (cdr L))))
        (cons (funcall g (car L)) x))))
```

the built in mapcar is more powerful.
It can map functions of 2 or more args

Built-in Map Car

```
→ (mapcar #'+ '(1 0 1 2 2)
              '(2 1 0 1 3)
              '(7 1 2 1 0)
              '(1 1 6 1 5))   =>   (11 3 9 5 10)

(mapcar #'cons '(A B (6 7) (D (3 4)))
               '((1 2) (3 4) (x y z) (w)))

  =>  ((A 1 2) (B 3 4) ((6 7) x y z) (D (3 4) w))
```

10/23/17

Monday, October 23, 2017 3:18 PM

102317

Finishing Off Lisp Material

Lambda Expressions : are used to create anonymous/nameless functions Example $2x^2+4x+7$ is an example; $f(x)=x^2+2x+7$ is a named function

To write a named function in Lisp: (defun fcn (+ (* x (* x x)) -2) 7))

(lambda (x) (+ (* x (* x x)) -2) 7))

Lambda Expression can be used in a number of ways:

1. lambda at beginning of list

((lambda (x) (+ (* x x) -2) 7) 3) => $3^3-2(3)+7 = 10$ // returns same result as (f 3)

((lambda (x y z) (+ (car x) y (car y))) '(1 2 3) 17 '(4 5)) => 2+17+4 = 23

(car x): get second element; (car y): first of y

2. A lambda expression may be passed as an argument when calling another function

Recall Sigma (sigma f m n) = $\sum_{i=m}^{n} f(i)$ Example (sigma #'(lambda (k) (* k k)) 3 6) => $\sum_{k=3}^{6} k^2 \Rightarrow 3^2+4^2+5^2+6^2 = 86$

Example: (mapcar #'(lambda (x y z) (+ (cadr x) y (car y))) '((1 2 3) (4 5) (6 7) (8 9 0)) → x values '(9 10 20 100) '((12 13) (25) (83) (234)))

using mapcar to map it will need 3 lists, bc there are 3 args

returned result → (24+12 5+10+25 7+20+8 9+100+2) = (19 40 35 111)

3. To create a function that will be returned as the result of another function

Will write a function compose as an example. Compose → (compose f g) → f∘g → (f∘g)(x) → f(g(x))

```
(defun compose (f g)
  (lambda (x) (funcall f (funcall g x))))
```

(funcall f ...): interpret the val of f as a function, not a variable

4. Importantly, a lambda expression may use any formal parameter of a function in whose body it occurs.

foo question

Example: Recall (power y n) => y^n The built-in version is called Expt (expt 2 4) = $4^2 = 16$

↳ want to sum

Let us use Expt function to create Sum-powers such that (sum-powers k m n) = $m^k+(m+1)^k+\dots+(n-1)^k+n^k$

```
(defun sum-powers (k m n)
  (sigma #'(lambda (x) (expt x k)) m n))
```

(sigma #'(lambda (x) (expt x k)) m n): the power function

(sum-powers 2 3 6) => $3^2+4^2+5^2+6^2 = 86$

Example Inc-list-2 from assignment 4

Recall (inc-list-2 '(2 7 3 1 9) 1000) → (1002 1007 1003 1001 1009)

```
(defun inc-list-2 (L n)
  (mapcar (lambda (k) (+ k n)) L))
```

K ← K+n

Apply

Apply is a generalization of funcall that allows args of the called function to be passed in a list

(apply #'+ '(2 3 1 6 5)) => 17

(apply #'+ a '(3 1 6 5))

⋮

(apply #'+ 2 3 1 6 5 NIL) => 17

#'append, #'list, #'+, #'*

(funcall #'+ 2 3 1 6 5) => 17 = (+ 2 3 1 6 5)

Recall from #4 Sum

(sum '(2 1 6 3)) = 2+1+6+3 = 12

Sum can be defined as such:

```
(defun sum (L)
  (apply #'+ L))
```

Relationship between Apply & Funcall

(funcall f e_1 e_2 ... e_k)

≡ (apply f (list e_1 e_2 ... e_k))

(apply f e_1 e_2 ... e_k NIL)

f a variable, like a formal parameter

(apply #'mapcar #'+ '((1 2 3) (4 5 6) (7 8 9) (0 1 0))) (1 2 3 4)

≡ (mapcar #'+ '(1 2 3) '(4 5 6) '(7 8 9) '(0 1 0)) => (1+4+7+0 2+5+8+1 3+6+9+0) = (12 16 18)

// here we are adding a 4x3 matrix

Syntax

When compiling or deconstructing a program, the first phase is to decompose the source text into a sequence of Tokens

Examples of Tokens in Java:

while
Identifier
+
-
)
(
unsigned int literal

A token: Is a set of strings of characters. Each of which is called an instance of that token

example 3 instances of Identifier: x apple y237

Many Token (eg)) have just one instance

10/25/17

Wednesday, October 25, 2017 3:09 PM

102517

Very Important !! Do the reading and Exercise you were asked to do.

Last Time : Spoke on Tokens e.g. Identifier, >= , ; , + , unsigned-int-literal

Instance Include: apple, x23 ; 1 instance each ; Instance4Thisslide :

An instance of a sequence of tokens $T_1, \ldots, T_n$ is a piece of text that the compiler or interpreter should decompose into a sequence of tokens $T_1, \ldots, T_n$

Identifier >= unsigned-int-literal

x >= 23

Next, we will define the concept of a syntactically valid X where X is a programming language construct such as: "Java while statement" or "C++ compilation unit"

```
x = y [17] / 34 ;      possibly legal
a = a[17] / 0   ;      illegal, but syntactically valid   // b/c the correct tokens line up correctly, in proper place
```

To do this we first define the concept of a syntactically valid sequence of tokens. The language designer is responsible for formulating a definition this concept such that:

- A sequence of Tokens $T_1, \ldots, T_n$ is syntactically valid if (and, roughly speaking, only if) some instance $T_1, \ldots, T_n$ is a possibly legal X
  - * possibly legal = legal in an appropriate arrangement

x = y [17] / 34 ;

Identifier = Identifier [unsigned-int-literal] / unsigned-int-literal ; ← A syntactically valid sequence of tokens for a Java assignment statement

→ Any instance of a syntactically valid sequence of tokens is then called a syntactically valid X.

An important part of the compiler/interpreter is to verify if the source program is syntactically correct → // The first phase of code checking

For most programming languages the only tokens that have more than one instance are Identifier & literals of various types e.g. unsigned-int-literal, unsigned-double-literal, charstring-literal

When X is a construct of such a language, we have that: A piece of text is syntactically valid X if (and roughly, only if) it can be obtained from a possibly legal X by making zero or more changes of the following kinds.

1. Replace an occurrence of identifier with another instance of Identifier
2. Replace an occurrence of a literal with another literal of the same type.

Syntax & Semantics of Expressions

Expressions can be written in many notations.
Let us consider the Java expression: f (g (h (1,2), f (3,4)), 5)

we can write this :

① Infix notation : each binary operator is written between its operands

((1 h 2) g (3 f 4)) f 5

② In an Infix notation with precedence classes, in which f & g have equal precedence and h has a higher precedence than f & g, and the precedence classes are left-associative

1 h 2 g (3 f 4) f 5

if f = +, g = −, h = * => 1 * 2 − (3 + 4) + 5 vs. (1 * 2) − (3 + 4) + 5

③ In Lisp Notation

(f (g (h 1 2) (f 3 4)) 5)

④ Prefix-Notation [= Lisp Notation without parentheses

f g h 1 2 f 3 4 5 // this isn't as bad if you know the arity of each function name

⑤ In "anti-Lisp" notation, which is like Lisp notation except function name is written after its arguments

(+ 3 4 5) ⟶ Anti = (3 4 5)

(f (g (h 1 2) (f 3 4)) 5) ⟶ (((1 2 H) (3 4 F) g) 5 f)

⑥ Post-fix (also called RPN → Reverse Polish Notation) or (I = "Anti-Lisp Notation w/o parentheses

(((1 2 H) (3 4 F) g) 5 f) → 1 1 2 H 3 4 F G 5 F

Infix Notation:

What is a Syntactically Valid infix notation expressions (SVIE) can be defined as follows

e is an SVIE if

① E is an instance of a Identifier

② E is (e_1) where e_1 is a SVIE

③ E is e_1 op e_2 where "op" is a binary operator and each e_1 & e_2 are SVIE

④ E is 'op' e_1 where "op" is a prefix unary operator & e_1 is an SVIE

⑤ E is e_1 'op' where "op" is a postfix unary operator & e_1 is an SVIE

As you can see from 1-5 only operators of arity > 2 are not allowed in Infix Notation

Moreover each unary operator needs to be classified as a prefix operator or a postfix

Warning:

In many cases, more than one of 3-5 applies to an SVIE and 3 may apply in more than one way as in:

a − b * c + d

possible choices of "op" for rule 3

10/30/17

103017

Monday, October 30, 2017 3:04 PM

Last Time: An SVIE (syntactically valid Infix Expression) is an expr of the following forms:

① E is an instance of a Identifier

② E is (e_1) where e_1 is a SVIE

③ E is e_1 op e_2 where "op" is a binary operator and each e_1 & e_2 are SVIE

④ E is 'op' e_1 where "op" is a prefix unary operator & e_1 is an SVIE

⑤ E is e_1 'op' where "op" is a postfix unary operator & e_1 is an SVIE

Semantics

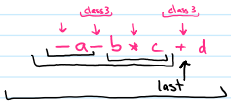
The semantics of an expr tells you how an expr should be evaluated.

- If e is an identifier or literal constant (case 1 ↑) then the val of e = the val of identifier or literal
- If e is (e_1) (case 2 ↑) then e.val = e_1.val
- Otherwise (case 3–5), let "op" be the operator that should be applied last ← Note: We will explain below precedence & associativity rules can be used to return which operator
  then if e is e_1 op e_2, then
  e.val = the result of applying op with e_1.val as 1st operand & e_2.val as 2nd operand

  if e is "op" e_1 or e_1 "op" then e.val = the value of applying "op" to e_1

	prefix	binary	associativity
Class 1	−		right
Class 2		*	left
Class 3		−+	left

− −x allowed ⇒ (− (−x))



(+) will be applied last, because of + is lowest precedence & left associativity.

Note: If it is possible that e.val has no meaning according to the above rules b/c of Semantic errors such as:

an uninitialized value, type mismatch, divided by 0 or (in a language that requires variable to be declared) → undeclared variable

How to determine which operator should be applied last

The precedence & associativity rules of an infix notation (which are specified by the designers of that notation) partition the operators in to ranked precedence classes & they specify for each class whether the class is left/right associative.

Step 0: Let e' = e w/o any of the surrounding parens () // example: e = ((−a−((b * c)+d))) then e' −a−((b*c)+d)

Step 1: Find all top-level operators (not in parens ()) of lowest precedence in e' // −a−((b*c)+d) then − & − are the top-level

→ with all the precedence classes binary "ops" is the lowest precedence of top level operators

Step 2: The "op" that should be applied last is the rightmost (left assoc) or leftmost (right assoc) "ops" found by step 1 according to left or right - associativity

Another Example

	binary	pre	post	associativity
Class 1		~		right
Class 2	+ −	+−		left
Class 3	& ∧ @			right
Class 4	# $			left

Find last applied op

A: (− x + z $ (~ y ∧ z) & a @ ~ z ∧ x # y − 1)

B: (− x + z @ (~ y # z) & a @ ~ z ∧ x @ y − 1)

Step 1 = locate lowest precedance (!)

Step 2 Assoc = left → look for rightmost = #
Assoc = right → look for leftmost = @

Warning: Precedence & Associativity rules do not (in general) uniquely determine which operator is applied first

Prefix & Postfix Notations

An expression E is a S.V. prefix notation expr if:

1. E is an identifier or a literal constant
2. E is "op" $e_1 \ldots e_K$ where op is a K-ary op

An expr E is a S.V. postfix notation expr if:

1. E is an identifier or literal constant
2. E is $e_1 \ldots e_K$ "op" where "op" is a K-ary op

2. E is "op" $e_1 \ldots e_k$ where op is a K-ary op

2 E is $e_1 \ldots e_k$ "op" where "op" is a K-ary op

Semantics

In case 2, e.val = the result of applying "op" to $e_1.val, \ldots, e_k.value$ with $e_i.val$ as the i^{th} operand of op

Differences from Infix:

- No parenthesis
- Operators have arity ≥ 2
- precedence classes are not used

Prefix & Postfix expr can be evaluated using a stack as follows:

- Read the expr left to right (postfix) right to left (prefix)
- when you see a variable or constant, push its value on to the stack.
- when you see a K-ary operator, pop $\frac{k}{2}$ values from stack & apply the operator the values
- then push result onto the stack

11/6/17

Monday, November 6, 2017 3:05 PM

Lexical Syntax (See 2.3)

The syntax of programming languages is specified in terms of:

① Tokens → (Examples: if, /,), (, *, +, unsigned Int literal)
→ if (identifier <= unsigned-int-literal) identifier ++ ; (10 tokens)

② A chosen set of language constructs (such as "if statement", "expression", "variable declaration" and "statement"
– Note: Some of these constructs may be special cases of the others

Two kinds of questions that syntax specification needs to answer are:

A) For each token, exactly what sequence of characters constitute a valid instance of that token?
- Note: For most tokens will have just one instance or a few instances. However tokens that = Identfier or -literal may have many instances.

B) For each of the chosen language constructs, exactly what sequences of tokens constitute a syntactically valid sequence of tokens
The part of the syntax specifications that answers A is called the lexical syntax spec. However the "main" part of sytax specification will relate to B

* Recall that the answer to B for a language construct X should have the following property:

– A sequence of Tokens $T_1, \ldots, T_n$ is syntactically valid if (and, roughly speaking, only if) some instance $T_1, \ldots, T_n$ is a possibly legal X
* possibly legal = legal in an appropriate arrangement

Identifier: An identifier is the name of a variable, function, class, etc.

Keyword: A token such as (If, while, do) that look like an identifier but has an entirely different role in programs.

Reserved Word: A token that look like an identifier but cannot be used as an Identifier is Reserved.

- In many languages (including C++ & Java) every Keyword is also a reserved word
- However Lisp keywords are not reserved words. Lisp has no reserved words. Any Lisp Keyword can be used as an identifier in the right context

Example:

```
(defun nil (if)                  > (nil 7)
(if ( > if 5)                      10
    (+ if 5)                     > (nil 3)
    (- if 2)))                     1
```

Is every reserved word a keyword? → Authors disagree on this. A language may have reserved words that do nothing (E.g. Goto, const) In Java

- As mentioned earlier, tokens that represent an Identfier or -literal constants of certain types have many instances. However, it should be noted that two different instances of the same token might be semantically equivalent.
For example: 0x7A & 122 are semantically equivalent instances of the "unsigned-int-literal" token

apple orange | advisor adviser | busses buses

3.10 3.1 3.1e0
- are semantically equivalent to the "unsigned-double-literal" token

DOG, Dog, and dog
- are semantically equivalent instances of the "identifier" token

- Most languages today are free-format languages. In a free-format language white space characters (blanks, tabs, newline) are ignored when they appear before a token

Example: while (x<3): ≡ while (x < 3);

In free-format languages the programmers has much flexibility in formatting code. However programmers have to read poorly formatted code

Context Free Grammars (See 2.4)

Context free grammars are one way to precisely specify a set of finite sequences of symbols [A set of finite sequences of symbols is sometimes called a formal language]

Context free grammars were introduced by Chomsky in the 1950's; he used them to specify the syntax of natural languages (such as English)
A few years later, Backus independently reinvented the same concept, and proposed that context-free grammers be used to specify the syntax of the lang Algol 60

Context free grammars are one way to precisely specify a set of finite sequences of symbols [A set of finite sequences of symbols is sometimes called a formal language]

Context free grammars were introduced by Chomsky in the 1950's; he used them to specify the syntax of natural languages (such as English)
A few years later, Backus independently reinvented the same concept, and proposed that context-free grammers be used to specify the syntax of the lang Algol 60
This proposal was adopted in the Algol 60 report.

The grammar notation used in the Algol 60 report is called BNF (Backus-Naur Form), however we will use the term more loosely to mean:

An commonly used notation for writing context-free grammar

CS 316 Lecture 18

Lecture #18

Wednesday, November 8, 2017 15:10

Median = 18.5
N = 35
"Good Scores" -> [>= 22 = 88%]
Aim for 6 hours per week
Do all reading assignments and homework exercises
Read lecture notes
Read first-da handout
Read all emails (check email on euclid using pine =>2 times each week
Attend all classes

- Context free grammars (sec 2.4)
  - Note: Sethi uses the term BNF to mean the grammar notation that was used in the algol60 report, but we (like many other authors today) will use the term more loosely, to mean any commonly used notation for writing grammars
    - Example: Figure 2.6 on p42 of sethi is an example of BNF in our sense (but not Sethi's sense)
    - Figure 2.10 on p46 of Sethi shows how the same grammar can be written in what Sethi would call BNF
    - Fig2.6 - this is BNF in our sense, not Sethi's
      - E ::= E + T
      - Some authors use an arrow instead of the ::=
      - E -> E + T
    - Fig2.10 - this is BNF in both senses
      - <Expression> ::= <expression> + <term>
  - Grammars are used to precisely specify sets of finite sequences of symbols
    - The symbols here are called the terminals of the grammar
    - Each of the specified sets is called a non-terminal of the grammar
    - One of the non-terminals is considered to be the "main" non-terminal
      - This non-terminal is called the starting nonterminal (or start-symbol or sentence symbol)
    - The set of finite sequences of terminals that is denoted by the starting nonterminal is called the language of (or language generated by) the grammar. We often think of the other nonterminals as "helping" nonterminals that have been defined so
  - "Helping" nonterminals that have been defined so they can be used to define the starting nonterminal
  - Two uses of grammars are:
    1. In lexical syntax specification to specify the set of sequences of characters that constitute a valid instance of a token T that has many instances [Here T maybe a token that represents an identifier or a literal constant of some type]
       1) In the above, the terminals of the grammar are characters
    2. To specify sets of syntactically valid sequences of tokens for language constructs (such as "While statement", "statement", or "compilation unit")
       1) In the above, the terminals of the grammar are tokens
    3. It is also comon to "compose" grammars of type 1 with a grammar of type 2 to produce a grammar whose terminals are characters and some nonterminals represent tokens that have many instances.
       1) Note it is also common to do 1 using regular expression notation or just using English sentences

- Example from textbook
  - Here is an example of 1, this example is essentially the same as the grammar in Fig. 2.3 on p.36 of Sethi. You should also read footnote on page 38
  - The following grammar specifies unsigned floating point literals in a single programming language
  - In this language, such literals have the form
    - d . . . . D DOT d . . . . D
      - Think dot as giant period, think of it as a decimal point
      - D ->D means one ore more digits, brackets underneath it
    - In our language
      - 43143.4112 is good
      - .4112 is bad
      - 43142. is bad
      - 4.31e+3 is bad
  - Unsigned floating point literal -> UFPL
    1. Ufpl ::= ip . f
       - Ip is the "integer part", f is a "fraction"
    2. Ip ::= d
       1) Ip ::= ip d //going to be erased in a minute, this is originally 3
       2) This is saying anything that is ip … well if 4162.413 where before dot is ip, after dot is f, then that is a ufpl
       3) 3 is an ip, 34 is an ip d, 343 is an ip d with 34 and d, so that is an ip as well
       4) So the reason we omit the 1) is because they both have the same lefthand side, as such we draw a single bar to the next line like so
    3. | ip d
       1) The above lines 2 and 3 can be rewritten as Ip ::= d | ip d
    4. F ::= d
    5. | d f
       1) So why is 3432 where 343 = 343 ip, 2 = 2 d a good decompoition? b/c 343 = ((343) * 10 + 2 ; ip ::= ip_1 d, ip.value = ip_1.value) * 10 + d
       2) The issue however lies with fractions

4. f ::= d
5. | d f
    1) So why is 3432 where 343 = 343 ip, 2 = 2 d a good decompoition? b/c 343 = ((343) * 10 + 2 ; ip ::= ip_1 d, ip.value = ip_1.value) * 10 + d
    2) The issue however lies with fractions
        a) 0.3432 = 0.3432)3 + 0.432)/10
            i) f::=df
            ii) f.value(d.value + f_1.value)/10
6. d::= 0

.....

15. d ::= 0

11/13/17

111317

Monday, November 13, 2017 3:08 PM

Audio recording started: 3:14 PM Monday, November 13, 2017

Tiny J Assignment 1 is now Available. Do the installation before Wednesday's Class / Read Assignment Document before Class
Read the examples + debugging hints document / Before Exam II do exercises & reading for syntax
Before the final, do assignment reading on C++ virtual functions

Last Time:

unsigned floating point literal grammar

ufpl ::= ip . f
(one or more digits) (one or more dig)
2313 . 262 → 2313.262

1. ufpl ::= ip . f
2. ip ::= d
3. | ip d
4. f ::= d
5. | df
6. d ::= 0
7. | 1
15. | 9

This grammar has 10 terminals : 0, 1, ..., 9, . / It has 4 non-terminals : ufpl, ip, f, d

A terminal is a undefined constant symbol. A non-terminal is a variable that denotes a set of finite sequences of terminals

[In the above grammar, we will see that ip denotes the set of all non-empty finite sequences of digits, and f denotes the same set]

The grammar above consists of 15 grammar rules called Productions
Each production has the form:

N ::= α
(N: a single non-terminal; α: a (possible empty) sequence of terminals and/or non-terminals)

Note | is the "alternative" symbol : It means "The left side of this production is the same as the left side of the previous production.

Grammar Notation is free format — whitespace characters (including \n) are ignored ip ::= d
| ip d → ip ::= d | ip d

N ::= α says "any α is an N"

Ex: d ::= 4 says "4 is a d"
ip ::= d says "any d is a ip"
ip ::= ip d says "if you take an ip and append a d to that ip then you will get another ip"

↳ Examples: 3 ∈ D ; 4 ∈ ip ; 43 ∈ ip (ip)(d) ; 434 ∈ ip (ip)(d) ; 4344 ∈ ip (ip)(d)
4 ∈ d

For this grammar ufpl is the starting non-terminal

- unless otherwise indicated, the starting non-terminal that appears on the left side of the very first production
- recall: that every non-terminal denotes a set of finite sequences of terminals. The set of sequences denoted by the starting non-terminal is called the language of (or language generated by) the grammar
  This set is regarded as the most important set specified by the grammar
- An empty grammar looks like N ::= <empty> <empty>, ε, and λ are 3 ways to represent the empty string.

Example of using <empty>: Suppose we want to mod the above grammar so the ufpl may have NO digit before the decimal point: → Replace (2) with ip ::= <empty>

Parse Trees

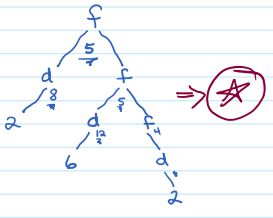
Question: Exactly which sequences of terminals belong to the set of sequences specified by a given non-terminal N

Answer: Use Parse Trees : A sequence of non-terminals T1, ..., Tn belongs to the set denoted by N if and only if T1, ..., Tn has a parse tree whose root is N.
Note: unless otherwise indicated, "parse tree" means "parse tree whose root is the starting non-terminal"

Example: Here is a parse tree which proves that the sequence of terminals 262 (2313.262) belongs to the set of sequences denoted by non-terminal f

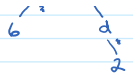
So here the root is f.

Parse tree of 262 whose root is f →



A parse tree with root N is an ordered rooted tree that has properties:

1. Each leaf is either a terminal or the symbol <empty>; an empty leaf has no children
2. Each non-leaf node is a non-terminal
3. The sequence of the children of a non-leaf node K is the right side of the production whose left side is the matching non-terminal
4. The root of the tree is the non-terminal N

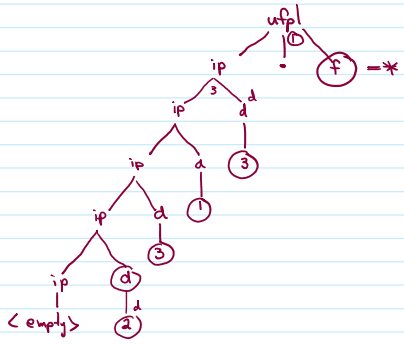


③ The sequence of the children of a non-leaf node K is the right side of the production whose left side is the matching non-terminal

④ The root of the tree is is the non-terminal N

A parse tree of a sequence of terminals $T_1, \ldots, T_k$ is the parse tree with the root N whose sequence of non-empty leaves in left to right order is $T_1, \ldots, T_k$

Here is a parse tree proving that 2312.262 belongs to the language by the modified grammar with ② as ip::= <empty>



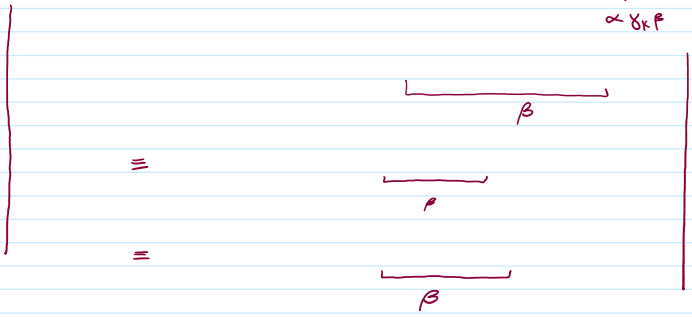
EBNF = Extended BNF
= BNF + (...) + [...] + {...}

Any set of sequences of terminals that can be specified in EBNF can be specified in BNF but EBNF is usually more concise & readable

Example 3

$N ::= \alpha (\gamma_1 \mid \ldots \mid \gamma_k) \beta$ = $N ::= \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \ldots \mid \alpha\gamma_k\beta$

$N ::= (\gamma_1 \mid \gamma_2) \mid \gamma_3 \rightarrow N ::= \gamma_1 \mid \gamma_2 \mid \gamma_3 \rightarrow N ::= (\gamma_1) \mid N ::= \gamma$



11/15/17

Wednesday, November 15, 2017 3:21 PM

[γ] = (γ | <empty>)

{ γ } = (<empty> | (γ) | (γ)(γ) | ...) = 0 or more

Translating EBNF Rules into BNF

The following Method always works. The idea is to produce a new non-terminal for each : (...) [...] {...}

{ α } β = A ::= β / αβ

β { α } = A ::= β / Aα

Example: Translate e ::= [+ | -] t { (+|-) t }

e ::= opt-sign t rest

opt-sign ::= + | - | <empty>

op ::= + | -

rest ::= <empty> | rest op t

ip ::= <empty> | ip d

defines ip as the set of 0 or more d's

11/27/17

Monday, November 27, 2017 3:15 PM

Audio Recording...

Audio recording started: 3:20 PM Monday, November 27, 2017

The TinyJ Compiler's Static and Stack-Dynamic Memory Allocation Rules

Static Memory Allocation for Static Variables in TinyJ:

The n^{th} static int or array reference variable in a TinyJ source file is given the data memory location whose address is $n-1$. (So the address of the first such variable is 0.) This rule ***does not apply to Scanner variables***: In TinyJ, Scanner variables are fictitious variables; no space is allocated to them.

Static Memory Allocation for String Literals in TinyJ:

The k^{th} string literal character in the source file is placed into the data memory location whose address is $m+k$, where m is the last address allocated to a static variable. (A difference between TinyJ and Java is that string literals in TinyJ are *not* objects. In Java the characters of string literals and other String objects are stored in the heap region of the JVM's memory, which is where the data fields of Java objects and elements of Java arrays are stored.)

Stackframes of Method Calls and How Locations Within Stackframes are Allocated:

Each time a method is called during program execution, a block of contiguous data memory locations known as the call's ***stackframe*** or ***activation record*** is allocated; this block of memory locations will be deallocated when the method returns to its caller. Each formal parameter* of the method and each local variable declared in the method's body will be allocated a location within that stackframe—see the allocation rules below. Each location within the stackframe is referred to by its ***offset*** relative to the stackframe location at offset 0. (If in a certain stackframe the data memory address of the location at offset 0 is 73, then the data memory address of the stackframe location at offset +5 is 73 + 5 = 78.)

Memory Allocation Rule for Local Variables Declared in TinyJ Method Bodies:

Whenever the compiler sees a declaration of a local variable (other than a Scanner variable) in the body of a method, that local variable is given the first stackframe location with offset ≥ +1 which has NOT already been allocated to another local variable *that is still in scope.* (So, ignoring Scanner variables, the stackframe offset of the ***first*** local variable in each method's body is +1.) **EXAMPLE**:

```
int func()
{           1  2    3
  int a, b[], c;              1, 2, 3
  ...
  if ( ... ) {
    int d, e[];               4, 5
    ...  4  5
  }
  else {
    int f, g;                 4, 5, 6
    ...  4  5
    int h;
    ...  6
  }
  ...
  int i;                      4
  ...  4
}
```

The offset of the lower functions are reset.

As in, if there was a function func_2, then the offsets within the function's variables would Be reset and reassigned according to the declarations within.

In this example: a gets offset 1 b gets offset 2 c gets offset 3 d gets offset 4 e gets offset 5
When f is declared, d and e are out of scope. So: f gets offset 4 g gets offset 5 h gets offset 6
When i is declared, f, g, and h are out of scope. So: i gets offset 4

Memory Allocation Rule* for Formal Parameters of TinyJ Methods:

Formal parameters are given locations with ***negative*** offsets; the last formal parameter of the method gets the stackframe location at offset −2, the second-last parameter gets the location at offset −3, etc. **EXAMPLE**: In a stackframe of any call of `int g(int p, int q[], int r)`, r gets offset −2, q gets offset −3, and p gets offset −4.

*This ***does not apply to main( )'s parameter***: In TinyJ—unlike Java—main( )'s parameter is ***not*** a real parameter. (**main( )'s stackframe has no locations with negative offsets.**)

Formal params (the params of a function) have negative offsets

int g (int p, int q[], int r)
−2 −3 −4

0, −1 are taken

Use of Offsets 0 and −1 [This subsection is relevant mainly to <u>TinyJ Assignment 3</u>.]

The stackframe locations at offsets 0 and −1 store information that is used to support return of control from a called method to its caller. Specifically:

In each stackframe other than main( )'s stackframe, the ***dynamic/control link*** is stored at offset 0. (In main( )'s stackframe, the location at offset 0 stores an implementation-dependent pointer.) In stackframes of methods other than main( ), the dynamic/control link is a pointer to the data memory location at offset 0 in the stackframe of the method's caller.

In each stackframe other than main( )'s stackframe, the ***return address*** is stored at offset −1. The return address is the ***code memory*** address of the next VM instruction to be executed after the current method returns control to its caller. (In main( )'s stackframe there's no location at offset −1.)

Allocation and Deallocation of Stackframes: An Example

Suppose a TinyJ program has methods `main`, `f()`, `g()`, `h()`, and this happens when it is executed:

(1) `main()` is called
(2) `main()` calls `f()`
(3) `f()` calls `g()`
(4) `g()` calls `h()`
(5) `h()` calls `f()`
(6) `f()` returns control to `h()`
(7) `h()` returns control to `g()`
(8) `g()` calls `f()`

Then stackframes are *allocated* in data memory at times (1), (2), (3), (4), (5), and (8); stackframes are *deallocated* at times (6) and (7). Thus there will be just 4 stackframes in data memory immediately after (8). Listed ***in order of increasing memory addresses***, these 4 stackframes will be:

the stackframe of `main()` allocated for call (1)
the stackframe of `f()` allocated for call (2)
the stackframe of `g()` allocated for call (3)
the stackframe of `f()` allocated for call (8)

Note that the stackframes of `h()` and `f()` allocated at times (4) and (5) would no longer exist: The stackframe of `f()` allocated at time (5) would have been deallocated at time (6), and the stackframe of `h()` allocated at time (4) would have been deallocated at time (7).

Comment on Scanner Variables

The data memory allocation rules for TinyJ variables do ***not*** apply to Scanner variables (such as the local variable `userInput` of `howManyRings()` in CS316ex2.java and the static variable `input` in CS316ex5.java). ***No memory at all is allocated for Scanner variables*** in TinyJ. A Scanner variable x in TinyJ can only be used in `x.nextInt()`. This is executed by reading an integer from the standard input stream `System.in` (which is usually associated with the keyboard) and returning its value. So the Scanner variable x is completely irrelevant. That is why TinyJ essentially ignores Scanner variables and never allocates memory for them. (In contrast, the Scanner variable x is ***not*** irrelevant when a ***Java*** program executes `x.nextInt()`: In Java, a Scanner object need not be associated with `System.in`—a Scanner object may, for example, be associated with any input file.) In TinyJ, the Scanner variable x in `x.nextInt()` is there only because we want TinyJ to be a subset of Java so that TinyJ programs will be compilable by a Java compiler.

Effects of Executing Each TinyJ Virtual Machine Instruction

"Push" and "pop" refer to the TinyJ VM's **expression evaluation stack** (the **EXPRSTACK**).

n denotes an arbitrary nonnegative integer — *addr* denotes an arbitrary code memory address
a denotes an arbitrary data memory address — *a'* denotes an arbitrary data memory address ≥ *a*
s denotes an arbitrary stackframe offset in the currently executing method activation's stackframe

If an assumption that is made by a VM instruction is ***not*** satisfied when that instruction is executed (e.g., if the item popped by LOADFROMADDR is not a pointer), then the effects of executing the instruction are unspecified.

TinyJ VM Instruction	Effects of Executing the Instruction
STOP	Halts the machine.
NOP	Does nothing.
DISCARDVALUE	Pops an item.
PUSHNUM *n*	Pushes the nonnegative integer value *n*.
PUSHSTATADDR *a*	Pushes a pointer to the data memory location whose address is *a*.
PUSHLOCADDR *s*	Pushes a pointer to the data memory location that is at offset *s* in the currently executing method activation's stackframe.
SAVETOADDR	Pops an item *v*. Pops an item *p*, which is assumed to be a pointer to a data memory location. Stores *v* in the memory location to which *p* points.
LOADFROMADDR	Pops an item *p*, which is assumed to be a pointer to a data memory location. Pushes the value that is stored in the memory location to which *p* points.
WRITELNOP	Writes a newline to the screen.
WRITEINT	Pops an item *i*, which is assumed to be an integer. Writes the integer *i* to the screen.
WRITESTRING *a a'*	Assumes that the data memory locations whose addresses are ≥ *a* but ≤ *a'* contain the characters of a string literal. Writes that string literal to the screen.
READINT	Assumes the character sequence of an int will be entered on the keyboard. Reads that character sequence and computes the int value it represents. Pushes that integer value.
CHANGESIGN	Pops an item *i*, which is assumed to be an integer. Pushes the value −*i*.
NOT	Pops an item *b*, which is assumed to be a Boolean value. Pushes the Boolean value NOT *b*.
op = ADD, SUB, MUL, DIV, MOD, EQ, LT, GT, NE, GE, or LE	Pops an item *i*, which is assumed to be an integer. Pops an item *j*, which is also assumed to be an integer. Pushes the integer or Boolean value *j* **op** *i*.

pushstataddr → for static variable, or vars from outside the current stackframe

pushlocaddr → for the var defined in the current running method

TinyJ VM Instruction	Effects of Executing the Instruction
op = AND or OR	Pops an item *b*, which is assumed to be a Boolean value. Pops an item *c*, which is also assumed to be a Boolean value. Pushes the Boolean value *c* **op** *b*.
JUMP *addr*	Loads *addr* into the program counter register.
JUMPONFALSE *addr*	Pops an item *b*, which is assumed to be a Boolean value. Loads *addr* into the program counter register if (and only if) *b* is **false**.
PASSPARAM	Allocates one location in the stack-dynamically allocated part of data memory. Pops an item and stores that item in the allocated location; it is expected that the item which is popped and stored will be the value of an actual argument of a method that is about to be called.
CALLSTATMETHOD *addr*	Allocates one location in the stack-dynamically allocated part of data

C SCI 316 (Kong): TinyJ Assignment 3

This assignment is to be submitted *no later than* ***Thursday, December 21***. [Even though this submission deadline is after the final exam, ***there will be questions on the final exam which test your ability to write code that needs to be written to complete this assignment***! Also, if me/hd unexpectedly goes down after 6 p.m. on the due date, the deadline will *not* be extended. Try to submit before noon than day, and on an earlier day if possible.]

See page 5 for the ***late*** submission deadline.

The assignment is to further develop your TJsm program so it will correctly ***execute*** the TinyJ virtual machine (VM) instructions that it generates. For each VM instruction, there will be a corresponding file **instr.java* in your *TJsam/virtualMachine* directory. Every such file has an `execute()` method. In *STOPinstr.java*, *ADDTOPFRinstr.java*, *HEAPALLOCinstr.java*, *WRITEINTEGERinstr.java*, and *READCHTinstr.java*, that method is written for you. In the other 29 **instr.java* files, the body of the `execute()` method contains a comment of the form */* ??????? */* ***which you need to replace with code that executes the corresponding VM instruction***! Your code must make appropriate changes to the expressions evaluation stack `CodeInterpreter.EXPRSTACK[]`, data memory [`TJ.data[]`], and VM registers (`CodeInterpreter.ESP_, PC_, FP_, ASP_` etc.), and must produce correct output (if any). See the example at the beginning of the "How to Do the Assignment" section below.

Some Information About the Implementation of the TinyJ Virtual Machine

In `CodeInterpreter.java`, the variables PC, IR, ESP, FP, and ASP (see lines 14 – 18) represent VM registers; the array EXPRSTACK[] (line 19) represents the expression evaluation stack.

PC contains the code memory address of the **next** instruction to be fetched for execution. (When that instruction is fetched, it is put into IR for execution.)

ESP stores a count of the number of items that are currently on the expression evaluation stack—if `ESP > 0`, then `EXPRSTACK[ESP – 1]` is the topmost element of the stack.

FP is a pointer to the data memory location at offset 0 in the currently executing method activation's stackframe. (Thus FP=k will be a pointer to the location at offset *k* in that stackframe.)

ASP is a pointer to the first unused location in the stack-dynamically allocated part of data memory—i.e., ASP points to the first location above the currently allocated locations in that part of data memory. ASP must therefore be increased/decreased by *k* when *k* stackframe locations are allocated/deallocated.

In *TJ.java*, the ArrayList `generatedCode` (line 30) and the array `data[]` (line 26) represent code and data memory, respectively. `TJ.generatedCode.get(a)` returns the instruction stored in code memory at address *a*. `TJ.data[a]` represents the data memory location whose address is *a*.

The VM's fetch-execute loop is in the `interpretCode()` method of `CodeInterpreter.java` (line 48); this is called by `start()` (line 247). Its most important lines are:

`75 IR = TJ.generatedCode.get(thePC = PC++);` and `77 IR.execute();`

Line 75 *fetches* the instruction stored (in code memory) at the address in PC and increments PC. Line 77 *executes* the instruction that has just been fetched.

> "The address of the instruction that is fetched is stored into *thePC*—so where that instruction is being executed, and immediately afterwards, *thePC* will contain the address of the instruction. When a debugging step occurs—see the "How to Debug If You are Stumped" section below—this VM uses the address in *thePC* to determine which instructions have just been executed. PC – 1 will usually be equal to *thePC*, but will sometimes be different because PC may be changed by execution of **JUMP**, **JUMPONFALSE**, **CALLSTATMETHOD**, or **RETURN**.

Installation of DEMO Instruction Classes

Installation of TinyJ Assignment 2 should have installed the instruction classes from my solution (the "DEMO instruction classes") into your *TJsol1asamu/TJsam/virtualMachine directory.*

Before you work on this assignment, use the following commands to copy these DEMO instruction classes into your *TJsam/virtualMachine* directory:

```
cp YJsol1asamu/TJsam/virtualMachine/*instr.class TJsam/virtualMachine
                  (on euclid and remus, working directory = your home directory)
copy /y YJsol1asamu\TJsam\virtualMachine\*instr.class TJsam\virtualMachine
                  (in a cmd.exe window on a PC, working directory = c:\316\java)
```

Then, since the DEMO classes have correct `execute()` methods, once you have completed TinyJ Assignment 2 you should be able to execute TinyJ programs (e.g., the 16-*CS316ex4*.java* files) as follows:

`java -cp . TJsam.TJ CS316ex0.java A.out`

E.g., `java -cp . TJsam.TJ CS316ex0.java A.out` should write the following to the screen:

```
Start the compiling DEMO instruction classes program.
PUSHSTATADDR PUSHFROMADDR LOADFROMADDR PUSHFRAMEADDR CALLSTATMETHOD ADD
 COMPARISON WRITEINT WRITELNOP WRITESTRING PUSHLOCADDR DUP SUB METHOD
```

(This says that your program is using my DEMO versions of those instruction classes, rather than your own versions of the classes.) The program will ask:

`Want debugging stop or post-execution dump? (y/n):`

Respond by typing `n` and the program will begin to execute the *CS316ex0* program. (The output should be the same as the output you get when you compile *CS316ex0.java* using `javac CS316ex0.java` and then enter `java CS316ex0` to execute the generated byte codes.)

How to Do the Assignment

This assignment assumes you have completed Assignment 2 and that you know what each TinyJ VM instruction should do when it is executed—if you are not clear as to what certain instructions should do when executed, refer back to the "Effects of Executing Each TinyJ Virtual Machine Instruction" pages at: **http://euclid.nmt.edu/~rlk/316/VM-instruction-set-and-hints-for-asn-2.pdf**

As an example of how to complete an `execute()` method, consider the `execute()` method of *ADDinstr.java*. As Java (unlike C++) is guaranteed to evaluate expressions from left to right, the */* ??????? */* in *ADDinstr.java* can be replaced with the following statement:

`EXPRSTACK[--ESP-1] += EXPRSTACK[ESP];`

After making this change, use the command

`javac -cp . TJsam/virtualMachine/ADDinstr.java`

to recompile *ADDinstr.java* and produce a file *ADDinstr.class* *which will replace the DEMO version of ADDinstr.class* you copied into TJsam/virtualMachine. The new version of *ADDinstr.class* can then be tested by executing a TinyJ program whose virtual machine code uses the ADD instruction. For example, you could enter

`java -cp TJsam.TJ CS316ex3.java 9.out`

When you do this you should find that ADD is no longer on the list of DEMO instruction classes reported by *TJsam.TJ*! [The *javac* and *java* commands above assume that your working directory is your home directory on venus or euclid, and *cs316/java* on a PC.]

I suggest you work on the other instruction classes in the following order. First, work on **MUL, SUB, DIV, MOD,** and **CHANGESIGN**, which are used by, e.g., *CS316ex0.java*. Next, work on **NOT, AND, OR, EQ, NE, LT, GT, LE, GE, PUSHNUM, WRITELNOP, WRITEINT, WRITESTRING, JUMP, JUMPONFALSE, PUSHSTATADDR, LOADFROMADDR,** and **SAVETOADDR**, which are used by, e.g., *CS316ex3.java*. Finally, work on **PUSHLOCADDR, PASSPARAM, CALLSTATMETHOD, INITSTKFRM,** and **RETURN**, all of which are used by, e.g., *CS316ex5.java*.

Each time you complete the `execute()` method of a file *xxrinstr.java*, it might well be a good idea to immediately compile that *.java* file and then (if there are no compilation errors) immediately test your `execute()` method by using *TJsam.TJ* to execute a *CS316ex4.java* program whose virtual machine code uses the instruction in question. If *TJsam.TJ* fails to execute that *CS316ex4* program correctly, then your new `execute()` method is bugged and you need to fix it. [If you want to work on another instruction's `execute()` method first, then copy my DEMO version of *xxrinstr.class* back from your *Y3sol1asamu/TJsam/virtualMachine* directory into your *TJsam/virtualMachine* directory, so it replaces your bugged version.] If the *CS316ex4* program is executed correctly, save a backup copy of the file that contains your new `execute()` method—e.g., put the copy on **venus** if you are working on **euclid**.

How to Debug If You are Stumped

Suppose that you have just written the `execute()` method of `LOADFROMADDRinstr.java` (and that any other `execute()` methods you may have written earlier all work). After compiling *LOADFROMADDRinstr.java* (`javac -cp . TJsam/virtualMachine/LOADFROMADDRinstr.java`), if your program *TJsam.TJ* executes some TinyJ input file [say, *CS316ex3.java*] incorrectly, try the following debugging method:

Run *TJsam.TJ* on that same input file again, ***but set the TinyJ virtual machine to stop as soon as it has executed a* `LOADFROMADDR` *instruction***. Do this as follows:

```
java -cp . TJsam.TJ CS316ex3.java 3a.out
Want debugging stop or post-execution dump? (y/n):  y
Enter MINIMUM no. of instructions to execute before debugging stop.
(Enter -1 to get a post-execution dump but no debugging stop.):  0
Stop after executing what instruction? (e.g., PUSHNUM)
(Enter * to stop after executing just 0 instructions.):  LOADFROMADDR
```

Next, ***copy my DEMO version*** of *LOADFROMADDRinstr.class* back from your *Y3sol1asamu/TJsam/virtualMachine* directory into your *TJsam/virtualMachine* directory, and then repeat the above with the same debugging stop settings. ***But this time use a different output file name***—3b.out instead of 3a.out, say. After these two runs, the file *3a.out* will contain dumps of the contents of the virtual machine's registers and memory locations ***immediately before and immediately after*** execution of ***your*** version of *LOADFROMADDRinstr*'s `execute()`, whereas *3b.out* will contain the same information for ***my*** version of *LOADFROMADDRinstr*'s `execute()`. Use **diff -c** or **fc /a** to compare *3a.out* and *3b.out*. Their dumps ***before*** execution of the **LOADFROMADDR** instruction should be identical. (If they differ, one of the `execute()` methods you wrote earlier is bugged!) But the dumps ***after*** execution of **LOADFROMADDR** may well be different, in which case you should be able to see what your version of *LOADFROMADDRinstr*'s `execute()` did wrong.

In the event that the dumps ***after*** execution of **LOADFROMADDR** are also identical, make a note of the total number of *instructions executed before execution was stopped*—this number will be shown near the bottom of the "after" dumps. If this number is, say, **27**, then repeat the above process, *but this time set the virtual machine to execute 27+1 = 28 instructions*, and to then stop after it executes any **LOADFROMADDR** instruction:

```
Want debugging stop or post-execution dump? (y/n):  y
Enter MINIMUM no. of instructions to execute before debugging stop.
(Enter -1 to get a post-execution dump but no debugging stop.):  28
Stop after executing what instruction? (e.g., PUSHNUM)
(Enter * to stop after executing just 28 instructions.):  LOADFROMADDR
```


Use of `CodeInterpreter.POINTERTAG`

In the TinyJ virtual machine, a pointer to *data memory* address *a* is represented by the integer `p = CodeInterpreter.POINTERTAG + a`, where `CodeInterpreter.POINTERTAG` is a constant equal to `0x7FFF0000 = 01111111 11111111 00000000 00000000`. (This depends on the fact that the machine's data memory addresses are small enough to be stored in the lower two bytes of an int.)

When executing **PUSHSTATADDR** *a*, remember to push the *pointer* corresponding to address *a*. In other words, remember to push `a + POINTERTAG`.

When executing **LOADFROMADDR**, remember that the topmost stack item is a *pointer*: If *p* is on top of the stack, then the data memory address to load from is `p - POINTERTAG`.

When executing **SAVETOADDR**, remember that the second item on the stack is a *pointer*: If *p* is the second item on the stack, then the data memory address to write-to is `p - POINTERTAG`.

When executing **CALLSTATMETHOD, INITSTKFRM, PASSPARAM,** and **RETURN**, remember that ASP is a pointer and refers to the data memory address `ASP - POINTERTAG`.

When executing **PUSHLOCADDR** *a*, remember that FP *already contains a pointer*, so you should **not** add `POINTERTAG` to `FP+a`. You also should **not** add `POINTERTAG` to FP before storing it

112717

Audio recording started: 3:29 PM Monday, December 4, 2017

More Old Exam Questions

1. [5 pts.] The language TeenyJ is defined like TinyJ except that the syntax of `<expr1>` is given by:

`<expr1> ::= DIGIT(DIGIT | new int '[' <expr1> ']' [ '[' ']']* |`

Suppose the Parser class you completed for TinyJ Assignment 1 is to be modified so that it will parse TeenyJ programs (instead of TinyJ programs). Show how you would complete the following parsing method for `<expr1>`. (No code generation is expected.)

```
private static void expr1() throws SourceFileErrorException
{
  TJ.output.printSymbol(VSymbol);
  TJ.output.println();
}
```


Solution to Problem 2.

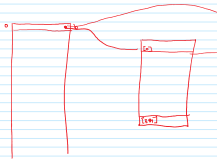
```
0:  PUSHSTATADDR   0
1:  PUSHNUM        315
2:  HEAPALLOC
3:  LOADFROMADDR
4:  LOADFROMADDR   1
5:  PUSHLOCADDR    0
6:  PUSHNUM        18
7:  PASSPARAM
8:  CALLSTATMETHOD 19
9:  SAVETOADDR
10: PUSHSTATADDR   0
11: LOADFROMADDR
12: PUSHNUM        171
13: ADD
14: SAVETOADDR
15: LOADFROMADDR   1
16: INITSTKFRM
17: WRITEINT       1   0
18: LOADFROMADDR   0
19: LOADFROMADDR
20: WRITELNOP      171
21: ADDTOPFR
22: LOADFROMADDR
23: WRITEINT
24: WRITELNOP
25: DUP
26: INITSTKFRM     0
27: PUSHLOCADDR    -2
28: LOADFROMADDR
29: PUSHNUM        5
30: LT
31: JUMPONFALSE    38
32: PUSHNUM        0
33: RETURN         1
34: PUSHLOCADDR    -2
35: LOADFROMADDR
36: PUSHNUM        1
37: SUB
38: PASSPARAM
39: CALLSTATMETHOD 26
40: POP
41: RETURN         1
```

2. [10 pts.] Complete the table below the following program to show the TinyJ virtual machine instructions that should be generated by TJsam.TJ (after completion of TinyJ Assignment 2) for this TinyJ program.

```
class Exercise
{
  static int b[] = new int[5][3];

  public static void main (String args[])
  {
    int i = a(19);
    b[2][1] = 11;
    System.out.print("a(19) is ");
    System.out.println(b[2][1]);
  }

  static int a(int n)
  {
    if (n < 3) return 3;
    else return n-a(n-1);
  }
}
```




```
0:  PUSHSTATADDR  0
1:  PUSHNUM  315
2:  HEAPALLOC
3:  PUSHNUM  19
4:  PASSPARAM
5:  CALLSTATMETHOD  46
6:  SAVETOADDR
7:  PUSHSTATADDR  0
8:  LOADFROMADDR
9:  PUSHNUM  271
10: ADDTOPFR
11: PUSHLOCADDR  1
12: PUSHNUM  3
13: MUL
14: PUSHSTATADDR  1
15: LOADFROM ADDR
16: SAVETOADDR
17: WRITEINT  1  0
18: PUSHSTATADDR  0
19: LOADFROMADDR
20: PUSHNUM  271
21: ADDTOPFR
22: LOADFROM ADDR
23: WRITEINT
24: STOP
25: INITSTKFRM  0
26: PUSHLOCADDR  -2
27: PUSHLOCADDR  -2
28: LOADFROMADDR
29: PUSHNUM  3
30: LT
31: JUMPONFALSE  35
32: PUSHNUM  0
33: RETURN  1
34: DUP  47
35: PUSHLOCADDR  -2
36: LOADFROMADDR
37: PUSHLOCADDR  -2
38: PUSHNUM  1
39: SUB
40: PASSPARAM
41: CALLSTATMETHOD  26
42: SUB
43: RETURN  1
```

Hint: Among the 40 instructions you are asked to write, there are 7 LOADFROMADDR instructions, 6 PUSHNUM instructions, 5 PUSHLOCADDR instructions, 2 each of the ADDTOPFR, CALLSTATMETHOD, INITSTKFRM, PASSPARAM, PUSHSTATADDR, SAVETOADDR, and SUB instructions, and 1 each of the HEAPALLOC, JUMPONFALSE, LT, RETURN, STOP, WRITEINT, WRITELNOP, and WRITESTRING instructions.

While reading this page, you should refer back when necessary to the pages of the Assignment 2 document which specify the effects of executing each VM instruction.

Comment on Problem 2 Regarding the Translation of the Statements
`b[2][1] = 11;` and `System.out.println(b[2][1]);`

Note: The **EXPRSTACK** column on the right shows the items on the expression evaluation stack immediately ***after*** each VM instruction has been executed. The stack grows downwards—when more than one item is on the stack the first line below the word **EXPRSTACK** refers to the ***bottom*** item on the stack.

`b[2][1] = 11;` is translated into the seven VM instructions that are shown on the left below. These instructions are put into code memory at addresses 80 – 86, as indicated on p. 1.

PUSHSTATADDR 0	Pushes **pointer to b**.	**EXPRSTACK** = **pointer to b**
LOADFROMADDR	Pops **pointer to b**. Pushes the **pointer to b[0] that is stored in b**'s location.	**EXPRSTACK** = **pointer to b[0]**
PUSHNUM 271	Pushes the **integer 271**.	**EXPRSTACK** = **pointer to b[0]** **271**
ADDTOPFR	Pops **271**, and **pointer to b[0]**. Pushes **(pointer to b[0]) + 271**, i.e., **pointer to b[271]**.	**EXPRSTACK** = **pointer to b[271]**
PUSHLOCADDR 1	Pushes **pointer to i**.	**EXPRSTACK** = **pointer to b[271]** **pointer to i**
LOADFROMADDR	Pops **pointer to i**. Pushes the **value stored in i**'s location (i.e., the **value of i**).	**EXPRSTACK** = **pointer to b[271]** **value of i**
SAVETOADDR	Pops **value of i**, and **pointer to b[271]**. Saves **value of i** into the location of **b[271]**.	**EXPRSTACK** = *is empty*

`System.out.println(b[2][1]);` is translated into the seven VM instructions that are shown on the left below. These instructions are put into code memory at addresses 18 – 24, as indicated on p. 1.

PUSHSTATADDR 0	Pushes **pointer to b**.	**EXPRSTACK** = **pointer to b**
LOADFROMADDR	Pops **pointer to b**. Pushes the **pointer to b[0] that is stored in b**'s location.	**EXPRSTACK** = **pointer to b[0]**
PUSHNUM 271	Pushes the **integer 271**	**EXPRSTACK** = **pointer to b[0]** **271**
ADDTOPFR	Pops **271**, and **pointer to b[0]**. Pushes **(pointer to b[0]) + 271**, i.e., **pointer to b[271]**.	**EXPRSTACK** = **pointer to b[271]**
LOADFROMADDR	Pops **pointer to b[271]**. Pushes the **value stored in b[271]**'s **location** (i.e., the **value of b[271]**.)	**EXPRSTACK** = **value of b[271]**
WRITEINT	Pops **value of b[271]**. Writes the value on the screen	**EXPRSTACK** = *is empty*
WRITELNOP	Writes a newline to the screen.	**EXPRSTACK** = *is empty*

Further problems to test your understanding:

3. Suppose we delete the line `static int b[] = new int[5][3];` from the TinyJ program of problem 2 but insert a line `int b[] = new int[3][6];` at the beginning of the body of `main`. (Thus `b` would become the first local variable of `main`, and `i` would become the second local variable of `main` rather than the first local variable.) How would the 14 instructions shown on the previous page change?
   `PUSHSTATADDR 0` would be changed to `PUSHLOCADDR 2`.
   Each occurrence of `PUSHSTATADDR 0` would be changed to `PUSHLOCADDR 1`.

4. Suppose that the first variable declaration in a certain TinyJ program is `static int b[][][];`. Suppose also that this variable `b` is used in the following statement later in the program: `System.out.println(b[3][5][1]);`. What TinyJ VM instructions would the TinyJ compiler translate the latter statement into? [**Note**: Although Exam 2 may have questions relating to arrays, Exam 2 will not have any questions such as this one that involves arrays of arrays. However, there may be a question on the *Final Exam* that involves arrays of arrays.]

Answer to problem 4, and explanation of the generated instructions:

```
PUSHSTATADDR 0   Pushes pointer to b.                                  EXPRSTACK = pointer to b
LOADFROMADDR     Pops pointer to b.                                    EXPRSTACK = pointer to b[3]
                 Pushes the pointer to b[3] that is stored in b's location.
PUSHNUM 5        Pushes the integer 5.                                 EXPRSTACK = pointer to b[3]
                                                                       5
ADDTOPFR         Pops 5, and pointer to b[3].                          EXPRSTACK = pointer to b[3][5]
                 Pushes (pointer to b[3]) + 5, i.e., pointer to b[3][5].
LOADFROMADDR     Pops pointer to b[3][5].                              EXPRSTACK = value of b[3][5]
                 Pushes the value stored in b[3][5]'s location (i.e., the value of b[3][5]).
PUSHNUM 1        Pushes the integer 1.                                 EXPRSTACK = value of b[3][5]
                                                                       1
ADDTOPFR         Pops 1, and pointer to b[3][5].                       EXPRSTACK = pointer to b[3][5][1]
                 Pushes (pointer to b[3][5]) + 1, i.e., pointer to b[3][5][1].
LOADFROMADDR     Pops pointer to b[3][5][1].                           EXPRSTACK = value of b[3][5][1]
                 Pushes the value stored in b[3][5][1]'s location (i.e., the value of b[3][5][1]).
WRITEINT         Pops value of b[3][5][1].                             EXPRSTACK = is empty
                 Writes the value on the screen.
WRITELNOP        Writes a newline to the screen.                       EXPRSTACK = is empty
```

JUMP *addr*	Loads *addr* into the program counter register.
JUMPONFALSE *addr*	Pops an item *b*, which is assumed to be a Boolean value. Loads *addr* into the program counter register if (and only if) *b* is **false**.
PASSPARAM	Allocates one location in the stack-dynamically allocated part of data memory. Pops an item and stores that item in the allocated location; it is expected that the item which is popped and stored will be the value of an actual argument of a method that is about to be called.
CALLSTATMETHOD *addr*	Allocates one location in the stack-dynamically allocated part of data memory; this will be the location at offset −1 in the callee's stackframe. Stores the program counter in the allocated location; the stored address is the call's return address. Loads *addr* into the program counter register.
INITSTKFRM *n*	Allocates one location in the stack-dynamically allocated part of data memory; this will be the location at offset 0 in the current method activation's stackframe. Stores the frame pointer in the allocated location; this will serve as the stackframe's dynamic/control link pointer. Loads a pointer to the allocated location into the frame pointer register. Allocates *n* more locations in the stack-dynamically allocated part of data memory; these will be the locations at offsets 1 through *n* in the current method activation's stackframe.
RETURN *n*	Assumes *n* is the number of parameters of the currently executing method. Assumes the location at offset 0 in the currently executing method activation's stackframe contains the dynamic/control link pointer. Assumes the location at offset −1 in the currently executing method activation's stackframe contains the return address. Loads the dynamic/control link pointer into the frame pointer register. Loads the return address into the program counter register. Deallocates the data memory locations that constitute the currently executing method activation's stackframe.
HEAPALLOC	Pops an item *i*, which is assumed to be a nonnegative integer. Allocates *i*+1 contiguous locations in the heap region of data memory; it is expected that the second through *i*+1st of those locations will be used to store the elements of an array of *i* elements. Stores in the first of the *i*+1 locations a pointer to the first location above the *i*+1 locations; the second through *i*+1st locations will all contain 0. Pushes a pointer to the second of the *i*+1 locations.
ADDTOPTR	Pops an item *i*, which is assumed to be a nonnegative integer. Pops an item *p*, which is assumed to be a pointer to the data memory location of the first element of an array *arr*. Pushes *p*+*i* (which is a pointer to the location of the array element *arr*[*i*], unless *arr* has ≤ *i* elements in which case an error is reported.

Example: The Tiny2 Compiler of Assignment 2 should translate the following Tiny2 source file into the Tiny3 VM instructions shown on the next page.

```
import java.util.Scanner;

class Simple1 {

  static Scanner myInput = new Scanner(System.in);
  static int x, y = 10;

  public static void main(String args[]){
    System.out.print("Enter num: ");
    x = myInput.nextInt();
    f(17, y, x-y);
    System.out.println(y + f(21,22,23));
  }

  static int f (int a, int b, int c){
    int v[] = new int[3];
    int g = 0;
    g(c, b + v[1], v);
    System.out.print("returning from f ... ");
    return y - a % b;
  }

  static void g (int d, int e){
    int x;

    y = d / e;
  }
}
```


```
Instructions Generated:

0:    PUSHSTATADDR    1                34:   INITSTKFRM      3
1:    PUSHNUM         10               35:   PUSHLOCADDR     3
2:    SAVETOADDR                       36:   PUSHSTATADDR    0
=================================      37:   LOADFROMADDR
3:    INITSTKFRM      0                38:   SAVETOADDR
4:    WRITESTRING     2    12          39:   PUSHLOCADDR     -3
5:    PUSHSTATADDR    0                40:   LOADFROMADDR
6:    READINT                          41:   PASSPARAM
7:    SAVETOADDR                       42:   PUSHLOCADDR     -3
8:    PUSHNUM         17               43:   LOADFROMADDR
9:    PASSPARAM                        44:   PUSHLOCADDR     3
10:   PUSHSTATADDR    1                45:   LOADFROMADDR
11:   LOADFROMADDR                     46:   ADD
12:   PASSPARAM                        47:   PASSPARAM
13:   PUSHSTATADDR    0                48:   CALLSTATMETHOD   68
14:   LOADFROMADDR                     49:   NOP
15:   PUSHSTATADDR    1                50:   WRITESTRING     13   33
16:   LOADFROMADDR                     51:   PUSHSTATADDR    1
17:   SUB                              52:   LOADFROMADDR
18:   PASSPARAM                        53:   PUSHLOCADDR     -4
19:   CALLSTATMETHOD   34             54:   LOADFROMADDR
20:   DISCARDVALUE                     55:   PUSHLOCADDR     3
21:   PUSHSTATADDR    1                56:   LOADFROMADDR
22:   LOADFROMADDR                     57:   MOD
23:   PUSHNUM         21               58:   SUB
24:   PASSPARAM                        59:   RETURN          3
25:   PUSHNUM         22               =================================
26:   PASSPARAM                        60:   INITSTKFRM      1
27:   PUSHNUM         23               61:   PUSHSTATADDR    1
28:   PASSPARAM                        62:   PUSHLOCADDR     -3
29:   CALLSTATMETHOD   34             63:   LOADFROMADDR
30:   ADD                              64:   PUSHLOCADDR     -2
31:   WRITEINT                         65:   LOADFROMADDR
32:   WRITELNOP                        66:   DIV
33:   STOP                             67:   SAVETOADDR
=================================      68:   RETURN          2
```

Code Generation Rules Used by the Tiny3 Compiler

1. The generated code begins with instructions which initialize each static int and static array reference variable that has an *explicit* initializer. [**Example**: The instructions at addresses 0 - 2 in the code generated for the **Simple1** source file.]

2. For variables that do **not** have an explicit initializer, no initialization code is generated. Static variables that are *not* explicitly initialized will have a value of 0 (in the case of static **int** variables) or **null** (in the case of static array reference variables) when code execution begins. In the Tiny3 VM, the data memory locations allocated to static variables all contain 0 when execution begins, and the **null** pointer is represented by 0.

5. Method bodies are translated in the order in which they appear. [**Example**: The code generated for main()'s body appears before the code generated for other methods' bodies.]

6. The code generated for each method (including main) starts with
   INITSTKFRM *n* (total number of stackframe locations needed for local variables declared in that method's body)
   [**Example**: The instructions in addresses 3, 34, and 68.]

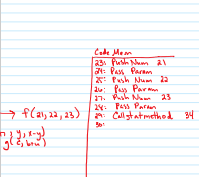
5. **main**()'s code ends with **STOP** [**Example**: The instruction at address 33.]

6. The code generated for each **void**-method (other than **main**()) ends with **RETURN** *k* Here *k* is the number of formal parameters that the method has. [**Example**: The instruction at address 68.]

7. A **return** *expression* ; statement in a method is translated into:
   <code which leaves the value of *expression* on top of EXPRSTACK>
   RETURN *k*
   Again, *k* is the number of formal parameters that the method has. [**Example**: The instructions generated for **return** y-a%b; at addresses 51 - 59.]

8. A method call `f(arg1, arg2, ..., argk)` within an *expression* is translated into:
   <code that leaves the value of arg1 on top of EXPRSTACK>
   PASSPARAM
   <code that leaves the value of arg2 on top of EXPRSTACK>
   PASSPARAM
   ...
   <code that leaves the value of argk on top of EXPRSTACK>
   PASSPARAM
   CALLSTATMETHOD <address of the first instruction in method f()'s code>
   [**Example**: The instructions generated for `f(21, 22, 23)` at addresses 23 - 29.]

9. A method call that is a standalone statement is translated to the same way as a method call within an expression, except that the **CALLSTATMETHOD** may be followed by **DISCARDVALUE POP**, or neither.
   (a) If the called method is known to return a value (either because it has already been declared to return a value, or because it has previously been called within an expression) then the **CALLSTATMETHOD** must be followed by **DISCARDVALUE** to pop the returned value off EXPRSTACK.
   (b) If the called method has already been declared as a void-method, then no **DISCARDVALUE** instruction is generated.



Video recording started: 3:42 PM Monday, December 4, 2017

data memory addresses are small enough to be stored in the lower two bytes of an int.]

When executing **PUSHSTATADDR** *a*, remember to push the *pointer* corresponding to address *a*. In other words, remember to push `a + POINTERTAG`.

When executing **LOADFROMADDR**, remember that the topmost stack item is a *pointer*: If *p* is on top of the stack, then the data memory address to load from is `p - POINTERTAG`.

When executing **SAVETOADDR**, remember that the second item on the stack is a *pointer*: If *p* is the second item on the stack, then the data memory address to write to is `p - POINTERTAG`.

When executing **CALLSTATMETHOD**, **INITSTKFRM**, **PASSPARAM**, and **RETURN**, remember that ASP is a pointer and refers to the data memory address `ASP - POINTERTAG`.

When executing **PUSHLOCADDR** *a*, remember that FP *already contains a pointer*, so you should ***not*** add POINTERTAG to FP+*a*. You also should ***not*** add POINTERTAG to FP before storing it during execution of **INITSTKFRM**, and should ***not*** subtract POINTERTAG when retrieving FP from the stackframe during execution of **RETURN**. Similarly, since FP and ASP both store pointers, do ***not*** add or subtract POINTERTAG when copying from ASP into FP or vice versa during execution of **INITSTKFRM** and **RETURN**.

When executing **WRITESTRING** *a b*, remember that *a* and *b* are ordinary data memory addresses, not pointers: Do ***not*** subtract POINTERTAG from *a* and *b*.

POINTERTAG is ***not used*** with code memory addresses: Never subtract POINTERTAG from PC, and never add POINTERTAG to a code memory address.

How to Test Your Solution

When you have correctly completed all the incomplete `execute()` methods, and recompiled each of the modified `*instr.java` files, test your program on the 16 `CS316exk.java` files. The runtime behavior should always be the same as when the same source file is compiled using `javac` and the resulting `.class` file is executed by the Java virtual machine. Also, the dump after execution of, say, 1000 instructions should be the same for your program as for my solution. Recall that you can run *my* solution as follows:

```
java -cp TJsolclasses:. TJasn.TJ CS316exk.java k.sol   [on euclid or venus]
java -cp TJsolclasses;. TJasn.TJ CS316exk.java k.sol   [on a PC]
```

How to Submit Your Solution

This assignment is to be submitted *no later than* ***Thursday, December 21***. [**Reminder**: Even though this deadline is after the final exam, ***there will be questions on the final exam which test your ability to write code that needs to be written to complete this assignment***. Also, if euclid unexpectedly goes down after **6 p.m.** on the due date, the deadline will ***not*** be extended. Try to submit before noon that day, and on an earlier day if possible.] See page 5 for the ***late*** submission deadline.

This assignment counts 2% towards your grade if the grade is computed using rule A. You may work with up to two other students; when two or three students work together, ***each student must submit separately***.

If you have been working on **euclid**, submit by following just the steps 1, 3, and 4 on the next page; do ***not*** do step 2! If you have been working on **venus** or your PC (***NOT euclid!***), do all of 1 – 4 on the next page. Your working directory should be your home directory on **venus** / **euclid** and `c:\316java` on a PC.

Page 4 of 7

1. Remove any incomplete `*instr.java` files you have ***NOT*** successfully completed from the `TJasn/virtualMachine` directory *of the machine you have been working on*. Then enter the following command on ***euclid***:
   `cp TJsolclasses/TJasn/virtualMachine/*instr.class TJasn/virtualMachine`
2. **This step does *not* apply if you have been working on euclid.** If, and ***only if***, you have been working on ***venus*** or your PC (***NOT euclid!***), enter the following command on ***euclid***:
   `rm TJasn/virtualMachine/*instr.java`
   *You must **not** do this if you have been working on euclid!* Then logout from **euclid**.

   If you have been working on **your PC**, you should now do 2(a) below. If you have been working on **venus**, do 2(b) below.

   2(a) If you have been working on **your PC**, create a jar archive of all the `*instr.java` files in `TJasn/virtualMachine`, as follows:
   `jar cvf instrs.jar TJasn/virtualMachine/*instr.java`
   This creates a jar archive file **`instrs.jar`** on your PC. Use an scp or sftp client to copy **`instrs.jar`** from your PC into your home directory on euclid. (If you have forgotten how to do this, see the second paragraph of the "**How to Submit Your Solutions**" section of the Lisp Assignment 3 document, but substitute **`instrs.jar`** for **`c:\316lisp\winston-3.lsp`** when following those instructions.)
   Then logon to **euclid** and extract the archived `*instr.java` files as follows:
   `euclid.cs.qc.cuny.edu> jar xvf instrs.jar`

   2(b) If you have been working on **venus**, enter
   `scp TJasn/virtualMachine/*instr.java xxxxx316@euclid.cs.qc.cuny.edu:TJasn/virtualMachine`
   on **venus**, where `xxxxx316` means your **euclid** username and the `:` after `.edu` is needed.
3. On **euclid**, recompile all the `*instr.java` files you have completed, as follows:
   `euclid> javac TJasn/virtualMachine/*instr.java`
   If any `*instr.java` file cannot be compiled on ***euclid***, you will receive no credit for this assignment. (In view of step 1 above, this should not happen.)
4. Test your program on **euclid**. (See the section "How to Test Your Solution" above.)

IMPORTANT: Do NOT open any of your submitted `*instr.java` files in an editor on euclid after the due date, unless you are resubmitting a corrected version of your solution as a *late* submission.

***Late* Submission Deadline**

I *expect* **euclid** to stay up after 6 p.m. on December 21. If **euclid** stays up, then ***late*** submissions of this and previous programming assignments (including corrected submissions) will be accepted until ***noon*** on ***Friday, December 22***; but this late submission deadline will ***not*** be extended if **euclid** goes down unexpectedly at any time after ***6 p.m. on December 21***. Note that if **euclid** goes down after 6 p.m. on Dec. 21 there is a possibility that it might not be brought back up until after noon on Dec. 22, in which case those students who did not submit before **euclid** went down may well not be able to submit at all! To avoid the risk of not being able to submit, make any late / corrected submissions no later than ***6 p.m. on Thursday, December 21***.

Page 5 of 7

Hints for TinyJ Assignment 3

Before you work on the assignment, carefully read page 1 above. When writing the `execute()` method for an instruction, if you are not clear as to what that instruction should do when it is executed, refer back to the "Effects of Executing Each TinyJ Virtual Machine Instruction" pages at:
http://euclid.cs.qc.cuny.edu/316/VM-instruction-set-and-hints-for-asn-2.pdf

Remarks on the `execute()` Methods You Have to Write

1. Instructions That Do Not Deal With Pointers to Data Memory Locations and Have No Operands

ADD, SUB, DIV, MUL, MOD, AND, OR

As explained on p. 2, **ADD**'s `execute()` method can be implemented as follows:

```
EXPRSTACK[--ESP-1] += EXPRSTACK[ESP];
```

The `execute()` methods for **SUB, DIV, MUL, MOD, AND,** and **OR** can be written analogously. Use 1 and 0 to represent **`true`** and **`false`**; then the execute() methods for **AND** and **OR** can use &= and |= where **ADD**'s execute() uses +=.

LE, GE, LT, GT, EQ, NE

One simple way to write the `execute()` methods for these instructions is to use conditional expressions. For example, **LE**'s `execute()` method could be implemented as follows:

```
EXPRSTACK[--ESP-1] = (EXPRSTACK[ESP-1] <= EXPRSTACK[ESP]) ? 1 : 0;
```

CHANGESIGN, NOT

These instructions should change `EXPRSTACK[ESP-1]` without changing ESP. (Note: `*= -1` is a concise way to negate an **int** variable; `^= 1` is a concise way to change its value from 1 to 0 or vice versa.)

WRITELNOP, WRITEINT

These instructions should print to `System.out`: **WRITELNOP** should print a newline, and **WRITEINT** should print an integer that is popped off EXPRSTACK. (Note that `EXPRSTACK[--ESP]` represents a value that is popped off EXPRSTACK.)

2. Instructions That Do Not Deal With Pointers to Data Memory Locations but Have Operands

The operand of any `OneOperandInstruction` (such as `PUSHNUMinstr` or `JUMPinstr`) is in its `operand` field (which it inherits from `OneOperandInstruction`). The two operands of a `WRITESTRINGinstr` are in its `firstOperand` and `secondOperand` fields (which it inherits from `TwoOperandInstruction`).

Recall from p. 1 that PC represents the VM's program counter: It stores the ***code memory*** address of the location from which the next instruction will be fetched for execution.

PUSHNUM 17	should push its operand onto EXPRSTACK (i.e., put 17 into `EXPRSTACK[ESP++]`).
JUMP 17	should put its operand 17 into PC, as PC represents the program counter.
JUMPONFALSE 17	should look at a value that is popped off EXPRSTACK (i.e., look at the value `EXPRSTACK[--ESP]`) and put its operand 17 into PC just if the popped value is 0.
WRITESTRING 7 17	should print (to `System.out`) the characters that are stored in the data memory locations with addresses ≥ its `firstOperand` 7 but ≤ its `secondOperand` 17. These locations are represented by: `TJ.data[7], TJ.data[8], ... , TJ.data[16], TJ.data[17]`. You must perform a (char) cast on the value in each location before printing it.

Page 6 of 7

3. Instructions That Deal With Pointers to Data Memory Locations

Before you write the `execute()` methods for these instructions, you should read the section "**Use of CodeInterpreter.POINTERTAG**" on p. 4.

Also, recall from p. 1 that FP ***stores a pointer to the data memory location at offset 0 in the currently executing method activation's stackframe***. (Thus FP + k will be a pointer to the location at offset k in that stackframe.)

PUSHSTATADDR 7	should push `7+POINTERTAG` (i.e., push a pointer to the data memory location whose address is 7) onto EXPRSTACK; in other words, it should put the pointer `7+POINTERTAG` into `EXPRSTACK[ESP++]`.
PUSHLOCADDR 7	should push `FP + 7` (i.e., push a pointer to the data memory location at offset 7 in the currently executing method activation's stackframe) onto EXPRSTACK; in other words, it should put the pointer `FP + 7` into `EXPRSTACK[ESP++]`.
LOADFROMADDR	If p is the pointer in `EXPRSTACK[ESP-1]`, execution of this instruction should copy the value in the data memory location to which p points (i.e., copy the value `TJ.data[p-POINTERTAG]`) into `EXPRSTACK[ESP-1]`; the copied value should overwrite the pointer p.
SAVETOADDR	If v is the value in `EXPRSTACK[ESP-1]` and p is the pointer in `EXPRSTACK[ESP-2]`, then this instruction should store v into the data memory location to which p points (i.e., it should store v into `TJ.data[p-POINTERTAG]`), and should also decrease ESP by 2 to pop v and p off EXPRSTACK.

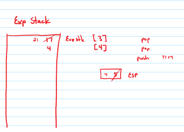
4. Instructions Associated with Calls of Static Methods and Return from Called Methods

Recall from p. 1 that ASP ***stores a pointer to the first unused location in the stack-dynamically allocated part of data memory***—i.e., ASP points to the ***first location above*** the currently allocated locations in that part of data memory.

So ASP must be increased / decreased by *k* when *k* stackframe locations are allocated / deallocated.

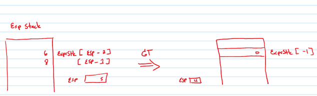
We will write	**S-PUSH** *y*	for	`TJ.data[ASP++ - POINTERTAG] = y;`
We will write	**S-POP** *y*	for	`y = TJ.data[--ASP - POINTERTAG];`

81: **PASSPARAM**	should	**S-PUSH** a value that is popped off EXPRSTACK —i.e., it should **S-PUSH** `EXPRSTACK[--ESP]`	
82: **CALLSTATMETHOD 671**	should	**S-PUSH** PC	[saves return addr (here, 83) into new frame]
	and then	Set PC to 671	[transfers control to the called method's code]
671: **INITSTKFRM 7**	should	**S-PUSH** FP	[saves caller's FP at offset 0 in the new frame]
	and then	Set FP to ASP-1	[makes FP point to offset 0 in the new frame]
	and then	Increase ASP by 7	[allocates space for callee's local variables]



public class Addinstr
{
void execute () throws VirtualMachineHaltException
{
-- ESP;
ExprStack [ESP -1] += ExprStack [Esp] ;
}
. . .
}

Greater than



main, and i would become the second local variable of `main` rather than the first local variable.) *How would the 14 instructions shown on the previous page change?*

Answer: `PUSHLOCADDR 1` would be changed to `PUSHLOCADDR 2`.
Each occurrence of `PUSHSTATADDR 0` would be changed to `PUSHLOCADDR 1`.

4. Suppose that the first variable declaration in a certain TinyJ program is `static int b[][][];` Suppose also that this variable `b` is used in the following statement later in the program: `System.out.print(b[7][29][5]);` What TinyJ VM instructions would the TinyJ compiler translate the latter statement into? [**Note**: Although Exam 2 may have questions relating to arrays, Exam 2 will ***not*** have any question such as this one that involves arrays of arrays. However, there may be a question on the ***Final Exam*** that involves arrays of arrays.]

Answer to problem 4, and explanation of the generated instructions:

`PUSHSTATADDR 0`	Pushes **pointer to b**.	**EXPRSTACK** ptr to b
`LOADFROMADDR`	Pops **pointer to b**. Pushes the **pointer to b[0] that is stored in b's location**.	**EXPRSTACK** ptr to b[0]
`PUSHNUM 7`	Pushes the **integer 7**.	**EXPRSTACK** ptr to b[0] 7
`ADDTOPTR`	Pops **7** and **pointer to b[0]**. Pushes **(pointer to b[0]) + 7** (i.e., **pointer to b[7]**).	**EXPRSTACK** ptr to b[7]
`LOADFROMADDR`	Pops **pointer to b[7]**. Pushes the **pointer to b[7][0] that is stored in b[7]'s location**.	**EXPRSTACK** ptr to b[7][0]
`PUSHNUM 29`	Pushes the **integer 29**.	**EXPRSTACK** ptr to b[7][0] 29
`ADDTOPTR`	Pops **29** and **pointer to b[7][0]**. Pushes **(pointer to b[7][0]) + 29** (i.e., **pointer to b[7][29]**).	**EXPRSTACK** ptr to b[7][29]
`LOADFROMADDR`	Pops **pointer to b[7][29]**. Pushes the **pointer to b[7][29][0] that is stored in b[7][29]'s location**.	**EXPRSTACK** ptr to b[7][29][0]
`PUSHNUM 5`	Pushes the **integer 5**.	**EXPRSTACK** ptr to b[7][29][0] 5
`ADDTOPTR`	Pops **5** and **pointer to b[7][29][0]**. Pushes **(pointer to b[7][29][0]) + 5** (i.e., **pointer to b[7][29][5]**).	**EXPRSTACK** ptr to b[7][29][5]
`LOADFROMADDR`	Pops **pointer to b[7][29][5]**. Pushes **value stored in b[7][29][5]'s location** (i.e., **value of b[7][29][5]**).	**EXPRSTACK** value of b[7][29][5]
`WRITEINT`	Pops **value of b[7][29][5]**. Writes this value to the screen.	**EXPRSTACK** *is empty*

Page 4 of 4

```
Solutions to the Recursive Descent Parsing Problem on Page 1
============================================================

First Solution:

  private void expr1() throws SourceFileErrorException
  {
    TJ.output.printSymbol(NTexpr1);
    TJ.output.incTreeDepth();

    if (getCurrentToken() == UNSIGNEDINT) {
        nextToken();
    }
    else if (getCurrentToken() == NEW) {
        nextToken();
        accept(INT);
        accept(LBRACKET);
        expr3();
        accept(RBRACKET);
        while (getCurrentToken() == LBRACKET) {
          nextToken();
          accept(RBRACKET);
        }
    }
    else throw new SourceFileErrorException("Malformed expression");

    TJ.output.decTreeDepth();
  }


Second Solution:

  private void expr1() throws SourceFileErrorException
  {
    TJ.output.printSymbol(NTexpr1);
    TJ.output.incTreeDepth();

    switch (getCurrentToken()) {
      case UNSIGNEDINT:
        nextToken();
        break;

      case NEW:
        nextToken();
        accept(INT);
        accept(LBRACKET);
        expr3();
        accept(RBRACKET);
        while (getCurrentToken() == LBRACKET) {
          nextToken();
          accept(RBRACKET);
        }
        break;

      default:
        throw new SourceFileErrorException("Malformed expression");
    }

    TJ.output.decTreeDepth();
  }
```

```
More Hand Translation Examples
------------------------------

[Note: You can also make up your own hand-translation examples: If
X.java is any valid TinyJ program, then the correct solution to the
problem of translating X.java can be obtained by running my solution to
TinyJ Assignment 2 with X.java as the input file.]

(a) Suppose Instruction.getNextCodeAddress() == 35 when a correct
    solution to TinyJ Assignment 2 begins to translate the following
    two methods.  What code is generated?

    static void m()
    {
      int x = 12, y = 9;
      System.out.print(p(17, y, x+5));
    }

    static int p (int a, int b, int c)
    {
      int u = a - b;
      return c - u;
    }

SOLUTION:
35:     INITSTKFRM        2
36:     PUSHLOCADDR       1
37:     PUSHNUM           12
38:     SAVETOADDR
39:     PUSHLOCADDR       2
40:     PUSHNUM           9
41:     SAVETOADDR
42:     PUSHNUM           17
43:     PASSPARAM
44:     PUSHLOCADDR       2
45:     LOADFROMADDR
46:     PASSPARAM
47:     PUSHLOCADDR       1
48:     LOADFROMADDR
49:     PUSHNUM           5
50:     ADD
51:     PASSPARAM
52:     CALLSTATMETHOD    55
53:     WRITEINT
54:     RETURN            0
55:     INITSTKFRM        1
56:     PUSHLOCADDR       1
57:     PUSHLOCADDR       -4
58:     LOADFROMADDR
59:     PUSHLOCADDR       -3
60:     LOADFROMADDR
61:     SUB
62:     SAVETOADDR
63:     PUSHLOCADDR       -2
64:     LOADFROMADDR
65:     PUSHLOCADDR       1
66:     LOADFROMADDR
67:     SUB
68:     RETURN            3
```

```
(b) An example involving arrays:

    class ArrayTest {
      static int b[] = new int[10];

      public static void main (String args[])
      {
        int a = 1;
        b[3] = a;
        System.out.println(b[3]+a);

        b = new int[5];

        int c[][] = new int [7][];
        c[4] = b;
      }
    }

    What would a correct solution to TinyJ Assignment 2 translate this into?

SOLUTION:
0:      PUSHSTATADDR      0
1:      PUSHNUM           10
2:      HEAPALLOC
3:      SAVETOADDR
4:      INITSTKFRM        2
5:      PUSHLOCADDR       1
6:      PUSHNUM           1
7:      SAVETOADDR
8:      PUSHSTATADDR      0
9:      LOADFROMADDR
10:     PUSHNUM           3
11:     ADDTOPTR
12:     PUSHLOCADDR       1
13:     LOADFROMADDR
14:     SAVETOADDR
15:     PUSHSTATADDR      0
16:     LOADFROMADDR
17:     PUSHNUM           3
18:     ADDTOPTR
19:     LOADFROMADDR
20:     PUSHLOCADDR       1
21:     LOADFROMADDR
22:     ADD
23:     WRITEINT
24:     WRITELNOP
25:     PUSHSTATADDR      0
26:     PUSHNUM           5
```

PASSPARAM

<code that leaves the value of arg_k on top of EXPRSTACK>
PASSPARAM
CALLSTATMETHOD <address of the first instruction in method f()'s code>
[**Example**: The instructions generated for `f(21, 22, 23)` at addresses 23 – 29.] → f(21, 22, 23)

9. A method call that is a *standalone statement* is translated in the same way as a method call within an expression, except that the **CALLSTATMETHOD** may be followed by **DISCARDVALUE**, **NOP**, or neither. → the f(n, y, x-y) or g(c, b+u)

(a) If the called method is known to return a value (either because it has already been declared to return a value, or because it has previously been called within an expression) then the **CALLSTATMETHOD** must be followed by **DISCARDVALUE** to pop the returned value off EXPRSTACK.

(b) If the called method has already been declared as a `void` method, then no **DISCARDVALUE** instruction is generated.

(c) If the called method has not yet been declared, and has not previously been called within an expression, then the compiler cannot tell if the method returns a value or not. In this case, the compiler essentially leaves a one-instruction gap after generating the **CALLSTATMETHOD** instruction. Later, when the compiler sees the declaration of the called method, it fills in the gap with either a **NOP** or a **DISCARDVALUE** instruction, according to whether the called method is declared to be a `void` method or a method that returns a value. [**Examples**: The instructions generated for `f(17, y, x-y)` at addresses 8 – 20, and the instructions generated for `g(c, b+u)` at addresses 39 – 49.]

25: Push Num 22
26: Pass Param
27: Push Num 23
28: Pass Param
29: Callstatmethod 34
30:

How Should Your Assignment 2 Compiler Translate TinyJ Source into TinyJ VM Code?

Answer: Your compiler should perform ***recursive descent translation***.

Notation For any non-terminal node *n* in the parse tree of a TinyJ source file, let *n*.code denote *the sequence of TinyJ VM instructions that should be generated by your compiler for the corresponding sequence of tokens* (i.e., the VM instructions that should be generated for the sequence of tokens that are the leaves of the subtree whose root is the node *n*).

Example If *n* is the <statement> node in the parse tree of the above program that corresponds to the statement `y = d / e;` in the body of the method `g`, then *n*.code = the instructions at code memory address 61 – 67. ← n.code produces

Example If *n* is the <expr2> node in the parse tree of the above program that corresponds to the expression `d / e` in the statement `y = d / e;`, then *n*.code = the instructions at code memory address 62 – 66.

When the recursive descent parser of TinyJ Assignment 1 is executed, there is one call of method *N*( ) for each occurrence *n* of a nonterminal <*N*> in the parse tree of the input file.) debug method might be on quiz

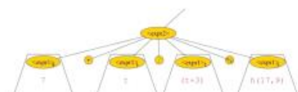
In Assignment 1, this call of *N*( ) reads the corresponding sequence of tokens and outputs the subtree of the parse tree whose root is *n* (and whose leaves are those tokens).

This is still true when the recursive descent translator of TinyJ Assignment 2 is executed, ***but in Assignment 2 the same call of N( ) also generates n.code.***

If *n* is an <expr *i*> node (where *i* = 1, 2, 3, 4, 5, 6, or 7), then:
n.code is code that leaves the value of the corresponding expression on top of **EXPRSTACK**.
(Recall that the TinyJ VM doesn't use registers for evaluating expressions but uses **EXPRSTACK** instead.)

Example Suppose *n* is an <expr2> node that corresponds to: `7 * t / (t+3) % h(17,9)` –
Recalling from Assignment 1 that the EBNF rule for <expr2> is
<expr2> ::= <expr1> {(*|/|%) <expr1>}
we see the subtree rooted at this <expr2> node has the following form (where the four trapezoids represent smaller subtrees): going over this

In this case, <expr2>.code = <expr1>$_a$.code
<expr1>$_b$.code
MUL
<expr1>$_c$.code
DIV
<expr1>$_d$.code
MOD



To make this same example more concrete, assume that in `7 * t / (t+3) % h(17,9)`
`t` is a static variable whose address is 5.
`h` is a method whose VM code begins at address 91 in code memory. – listen now

Then:

<expr2>.code = <expr1>$_a$.code	——	PUSHNUM	7
<expr1>$_b$.code	——	PUSHSTATADDR	5
		LOADFROMADDR	
MUL	——	MUL	
<expr1>$_c$.code	——	PUSHSTATADDR	5
		LOADFROMADDR	
		PUSHNUM	3
		ADD	
DIV	——	DIV	
<expr1>$_d$.code	——	PUSHNUM	17
		PASSPARAM	
		PUSHNUM	9
		PASSPARAM	
		CALLSTATMETHOD	91
MOD	——	MOD	

How the Compiler Can Generate a VM Instruction

There is a class in the `TJasn.virtualMachine` package for each kind of instruction.
To generate an instruction, create a new instance of that class; if the instruction has one or two operands, pass those operands as arguments when invoking the constructor. Examples:

`new WRITEINTinstr();`	generates	**WRITEINT**
`new JUMPONFALSEinstr(31);`	generates	**JUMPONFALSE 31**
`new WRITESTRINGinstr(21, 27);`	generates	**WRITESTRING 21 27**

The generated instruction is put in the next available location in code memory (which is represented by the ArrayList `TJ.generatedCode`).

Each time an instruction is generated, a line that reports this is written to the output file. For example:

```
*** Generating: 35:      PUSHLOCADDR      -1
```

How the Method `expr2()` in Assignment 1's `Parser.java` was Modified to Produce the Method `expr2()` in Assignment 2's `ParserAndTranslator.java`

Here is `expr2()` in Assignment 1's `Parser.java`, which is based on the EBNF rule
<expr2> ::= <expr1> {(*|/|%)<expr1>}

```
374        private static void expr2() throws SourceFileErrorException
375        {
376          TJ.output.printSymbol(NTexpr2);
377          TJ.output.incTreeDepth();
378
379          expr1();
380
381          while (   getCurrentToken() == TIMES
382                 || getCurrentToken() == DIV
383                 || getCurrentToken() == MOD) {
384
385            nextToken();
386
387            expr1();
388          }
389
390          TJ.output.decTreeDepth();
391        }
```

Consider the generation of <expr2>.code in the above example:

Recall that
<expr2>.code = <expr1>$_a$.code <expr1>$_b$.code **MUL** <expr1>$_c$.code **DIV** <expr1>$_d$.code **MOD**

Now if we just copy lines 379 – 388 above into Assignment 2's `expr2()` method, execution of those lines will generate: <expr1>$_a$.code <expr1>$_b$.code <expr1>$_c$.code <expr1>$_d$.code

[Line 379 generates <expr1>$_a$.code; line 387 generates <expr1>$_j$.code at the *j*th iteration of the while loop!]

So, to produce a correct version of `expr2()` that generates all of <expr2>.code, *we only need to add statements that generate the* **MUL**, **DIV**, *and* **MOD** *instructions*:

```
expr1();
while (   getCurrentToken() == TIMES
       || getCurrentToken() == DIV
       || getCurrentToken() == MOD) {
  Symbols op = getCurrentToken();          // ADDED
  nextToken();
  expr1();
  if (op == TIMES) new MULinstr();         // ADDED
  else if (op == DIV) new DIVinstr();      // ADDED
  else new MODinstr();                     // ADDED
}
```

After reading this page, you should be able to fill in the /* ???????? */ gaps on lines 638 through 682!

We will write **S-POP** *y* for *y* = *T3*.data[--*ASP* - *POINTERTAG*];

671: **PASSPARAM** should **S-PUSH** a value that is popped off *EXPRSTACK*
—i.e., it should **S-PUSH** *EXPRSTACK*[--*ESP*]

672: **CALLSTATMETHOD 671** should **S-PUSH** *PC* [saves return-addr (here, 83) into new frame]
and then Set *PC* to 671 [transfers control to the called method's code]

671: **INITSTKFRM 7** should **S-PUSH** *FP* [saves caller's FP as offset 0 in the new frame]
and then Set *FP* to *ASP*-1 [makes *FP* point to offset 0 in the new frame]
and then Increase *ASP* by 7 [allocates space for callee's local variables]

713: **RETURN 4** should Set *ASP* to *FP*+3 [deallocates space used by callee's variables]
and then **S-POP** *FP* [restores caller's *FP*]
and then **S-POP** *PC* [pops the saved return address into *PC*]
and then Decrease *ASP* by 4 [deallocates space used by formal parameters]


```
13:     LOADFROMADDR
14:     SAVETOADDR
15:     PUSHSTATADDR    0
16:     LOADFROMADDR
17:     PUSHNUM         3
18:     ADDTOPTR
19:     LOADFROMADDR
20:     PUSHLOCADDR     1
21:     LOADFROMADDR
22:     ADD
23:     WRITEINT
24:     WRITELNOP
25:     PUSHSTATADDR    0
26:     PUSHNUM         5
27:     HEAPALLOC
28:     SAVETOADDR
29:     PUSHLOCADDR     2
30:     PUSHNUM         7
31:     HEAPALLOC
32:     SAVETOADDR
33:     PUSHLOCADDR     2
34:     LOADFROMADDR
35:     PUSHNUM         4
36:     ADDTOPTR
37:     PUSHSTATADDR    0
38:     LOADFROMADDR
39:     SAVETOADDR
40:     STOP
```

(c) Example involving a while loop:

```
class Fall02c {
  static int a[] = new int[10];

  public static void main (String args[])
  {
      int x = 100;

      while (x > 10)
        x = f(x, 2);
  }

  static int f(int m, int n)
  {
      a[3] = m / n;
      return a[3];
  }
}
```

What would a correct solution to Tiny3 Assignement 2 translate this into?

```
SOLUTION:
0:      PUSHSTATADDR    0
1:      PUSHNUM         10
2:      HEAPALLOC
3:      SAVETOADDR
4:      INITSTKFRM      1
5:      PUSHLOCADDR     1
6:      PUSHNUM         100
7:      SAVETOADDR
8:      PUSHLOCADDR     1
9:      LOADFROMADDR
10:     PUSHNUM         10
11:     GT
12:     JUMPONFALSE     22
13:     PUSHLOCADDR     1
14:     PUSHLOCADDR     1
15:     LOADFROMADDR
16:     PASSPARAM
17:     PUSHNUM         2
18:     PASSPARAM
19:     CALLSTATMETHOD  23
20:     SAVETOADDR
21:     JUMP            8
22:     STOP
23:     INITSTKFRM      3
24:     PUSHSTATADDR    0
25:     LOADFROMADDR
26:     PUSHNUM         3
27:     ADDTOPTR
28:     PUSHLOCADDR     -1
29:     LOADFROMADDR
30:     PUSHLOCADDR     -2
31:     LOADFROMADDR
32:     DIV
33:     SAVETOADDR
34:     PUSHSTATADDR    0
35:     LOADFROMADDR
36:     PUSHNUM         3
37:     ADDTOPTR
38:     LOADFROMADDR
39:     RETURN          2
```

(d) Another example involving a while loop:

```
class Fall02b {

  static int a[] = new int [25];

  public static void main (String args[])
  {
      a[5] = 900;
      System.out.print(g(7));
  }

  static int g(int m)
  {
      int i = a[5];

      while (i > 10)
        i = i / m;

      return i;
  }
}
```

What would a correct solution to Tiny3 Assignment 2 translate this into?

```
SOLUTION:
0:      PUSHSTATADDR    0
1:      PUSHNUM         25
2:      HEAPALLOC
3:      SAVETOADDR
4:      INITSTKFRM      0
5:      PUSHSTATADDR    0
6:      LOADFROMADDR
7:      PUSHNUM         5
8:      ADDTOPTR
9:      PUSHNUM         900
10:     SAVETOADDR
11:     PUSHNUM         7
12:     PASSPARAM
13:     CALLSTATMETHOD  16
14:     WRITEINT
15:     STOP
16:     INITSTKFRM      1
17:     PUSHLOCADDR     1
18:     PUSHSTATADDR    0
19:     LOADFROMADDR
20:     PUSHNUM         5
21:     ADDTOPTR
22:     LOADFROMADDR
23:     SAVETOADDR
24:     PUSHLOCADDR     1
25:     LOADFROMADDR
26:     PUSHNUM         10
27:     GT
28:     JUMPONFALSE     37
29:     PUSHLOCADDR     1
30:     PUSHLOCADDR     1
31:     LOADFROMADDR
32:     PUSHLOCADDR     -2
33:     LOADFROMADDR
34:     DIV
35:     SAVETOADDR
36:     JUMP            24
```

```
37:     PUSHLOCADDR     1
38:     LOADFROMADDR
39:     RETURN          1
```

(e) The next example involves if as well as while:

```
import java.util.Scanner;

class Spring09 {

  static Scanner input = new Scanner(System.in);
  static int x;

  public static void main (String args[])
  {
    int a;

    x = input.nextInt();
    if (x > 1  &&  x < 20000) {
      while (x <= 20000) {
        int b = 2;
        x = x * 2;
      }

      int c = x;
      System.out.print(c);
    }
  }
```

What would a correct solution to Tiny3 Assignement 2 translate this into?

SOLUTION:

Note that the local variables b and c both have a stackframe offset of 2. At the point where c is declared, b no longer exists--b's scope is confined to the body of the while loop. Thus stackframe offset 2 can be reallocated to c.

```
0:      INITSTKFRM      2
1:      PUSHSTATADDR    0
2:      READINT
3:      SAVETOADDR
4:      PUSHSTATADDR    0
5:      LOADFROMADDR
6:      PUSHNUM         1
7:      GT
8:      PUSHSTATADDR    0
9:      LOADFROMADDR
10:     PUSHNUM         20000
11:     LT
12:     AND
13:     JUMPONFALSE     36
14:     PUSHSTATADDR    0
15:     LOADFROMADDR
16:     PUSHNUM         20000
17:     LE
18:     JUMPONFALSE     29
19:     PUSHLOCADDR     2
20:     PUSHNUM         2
21:     SAVETOADDR
22:     PUSHSTATADDR    0
23:     PUSHSTATADDR    0
```

listen now

Audio Recording...

Audio recording started: 3:12 PM Wednesday, November 29, 2017

Hints Relating to the Gaps on Lines 627, 723, and 593 in ParserAndTranslator.java

The Method `printArgument()` **[gap on line 627]**

The relevant EBNF rule is `<printArgument> ::= CHARSTRING | <expr>`

(a) In the case `<printArgument> ::= <expr>` the code to be generated is given by

```
<printArgument>.code = <expr>.code
                       WRITEINT
```

Assuming you correctly filled in the gap in the method `printArgument()` in Assignment 1, if you copy just that code into the body of Assignment 2's `printArgument()` then its call of `expr3()` will generate `<expr>`.code. To complete the `printArgument()` method, you would also need to insert a `new WRITEINTinstr();` statement.

(b) In the case `<printArgument> ::= CHARSTRING` the code to be generated is given by

```
<printArgument>.code = WRITESTRING a b
```

where a and b are the data memory addresses of the first and last characters of the CHARSTRING string literal that is to be printed. The `WRITESTRING a b` instruction can be generated by `new WRITESTRINGinstr(a,b);` with the appropriate addresses a and b; but how can your code find the two addresses a and b?

The solution is provided by the lexical analyzer: When `LexicalAnalyzer.nextToken()` sets `LexicalAnalyzer.currentToken` to CHARSTRING, it also sets the private variables `LexicalAnalyzer.startOfString` and `LexicalAnalyzer.endOfString` to the addresses of the memory locations where the first and last characters of the CHARSTRING will be placed. **`LexicalAnalyzer.getStartOfString()`** and **`LexicalAnalyzer.getEndOfString()`** are public accessor methods that return the two addresses.

listen now

The Method `expr1()` **[gap on line 723]**

The relevant EBNF rule is

```
<expr1> ::= '(' <expr> ')' | (+|-|!) <expr1> | UNSIGNEDINT | null
          |   new int '[' <expr3> ']' { '[' ']' }
          |   IDENTIFIER ( . nextInt '(' ')' | [<argumentList>]{'[' <expr> ']'} )
```

The `null` and the `IDENTIFIER ( . nextInt '(' ')' | [<argumentList>]{'[' <expr3> ']'} )` cases have been done for you in `ParserAndTranslator.java`. Here are hints for the other cases:

(a) In the case `<expr1> ::= '(' <expr7> ')'` the code to be generated is given by

```
<expr1>.code = <expr7>.code
```

Similarly, in the case `<expr1> ::= + <expr1>₁` the code to be generated is given by

```
<expr1>.code = <expr1>₁.code
```

In these two cases, assuming you correctly completed the body of the method `expr1()` when doing Assignment 1, if you use that code as the body of Assignment 2's `expr1()` then in the first case the call of `expr7()` will generate `<expr7>`.code, and in the second case the recursive call of `expr1()` will generate `<expr1>₁`.code.

(b) In the case `<expr1> ::= - <expr1>₁` the code to be generated is given by

```
<expr1>.code = <expr1>₁.code
               CHANGESIGN   ←
```

just need to the instr under the code from 1

Similarly, in the case `<expr1> ::= ! <expr1>₁` the code to be generated is given by

```
<expr1>.code = <expr1>₁.code
               NOT   ←
```

These two cases are similar to the second case of (a), except that you need to insert a `new CHANGESIGNinstr();` or a `new NOTinstr();` statement.

(c) In the case `<expr1> ::= UNSIGNEDINT` the code to be generated is given by

```
<expr1>.code = PUSHNUM v
```

←

where v is the numerical value of the UNSIGNEDINT integer literal. The `PUSHNUM v` instruction can be generated by `new PUSHNUMinstr(v);` with the appropriate value v; but how can your code find the value v?

The solution is provided by the lexical analyzer: When `LexicalAnalyzer.nextToken()` sets `LexicalAnalyzer.currentToken` to UNSIGNEDINT, it also sets the private variable `LexicalAnalyzer.currentValue` to the numerical value of the UNSIGNEDINT integer literal. **`LexicalAnalyzer.getCurrentValue()`** is a public accessor method that returns this value.

explanation is given. look at code for current Token

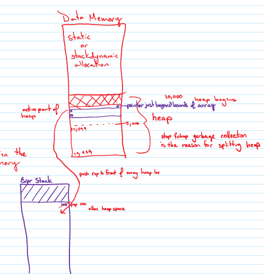
(d) In the case `<expr1> ::= new int '[' <expr3> ']' { '[' ']' }` the code to be generated is given by

```
<expr1>.code = <expr3>.code
               HEAPALLOC
```

ex: new int [100] or new int [ ][ ] ... an array of array references

Eval of: new int [100] must create new array object in the heap region of data memory

Instr: pushnum 100 heapalloc



Assuming you correctly completed the body of `expr1()` when doing Assignment 1, if you use that code as the body of Assignment 2's `expr1()` then `<expr3>`.code will be generated by a call of `expr3()`. You would need to insert a `new HEAPALLOCinstr();` statement.

back to here now

The Method `whileStmt()` **[gap on line 593]**

The relevant EBNF rule is

```
<whileStmt> ::= while '(' <expr7> ')' <statement>
```

and the code to be generated is given by:

```
<whileStmt>.code = a: <expr7>.code
                      JUMPONFALSE b
                      <statement>.code
                      JUMP a
                   b:
```

In `Instruction.java`, the static variable `Instruction.nextCodeAddress` is used to hold the code memory address of the ***next*** instruction to be generated. [This variable is incremented by 1 (by the constructor for the `Instruction` class) each time an instruction is generated.]

A call of `Instruction.getNextCodeAddress()` returns `Instruction.nextCodeAddress`. → get the val of A

Before calling `expr7()` to read the `<expr7>` expression and generate `<expr7>`.code, `whileStmt()` needs to call `Instruction.getNextCodeAddress()` and save the code memory address that is returned in an `int` local variable (which needs to be declared). This saved address will be needed as the operand a of the

```
JUMP a
```

instruction which `whileStmt()` must generate later!

`Instruction.OPERAND_NOT_YET_KNOWN` (line 13 in `Instruction.java`) is a constant integer that the compiler uses to represent an unknown operand value.

When `whileStmt()` begins to generate `JUMPONFALSE b` ***it does not know what the address*** b ***will be***. So at that time `whileStmt()` should merely generate

```
JUMPONFALSE Instruction.OPERAND_NOT_YET_KNOWN
```

and also ***save a reference*** to this JUMPONFALSE instruction in a local variable, to allow the instruction to be found again when `whileStmt()` is ready to fix up its operand:

```
JUMPONFALSEinstr jInstr = new JUMPONFALSEinstr(Instruction.OPERAND_NOT_YET_KNOWN);
```

```
24:     LOADFROMADDR
25:     PUSHNUM         5
26:     MUL
27:     SAVETOADDR
28:     JUMP            14
29:     PUSHLOCADDR     2
30:     PUSHSTATADDR    0
31:     LOADFROMADDR
32:     SAVETOADDR
33:     PUSHLOCADDR     2
34:     LOADFROMADDR
35:     WRITEINT
36:     STOP
```

(f) Suppose Instruction.getNextCodeAddress() == 45 when a correct solution to Tiny3 Assignment 2 begins to translate the following method, and suppose z is a static variable with address 2. What Tiny3 VM instructions would this method be translated into?

```
static int p(int x)
  {
   int y = 3, z;

   x = z + y;

   if (x < 10) z = x;
   else z = y;

   while (z <= 100) {
     System.out.println(z);
     z = z + y;
   }

   return z - x;
  }
```

```
SOLUTION:
45:     INITSTKFRM      2
46:     PUSHLOCADDR     1
47:     PUSHNUM         3
48:     SAVETOADDR
49:     PUSHLOCADDR     -2
50:     PUSHSTATADDR    2
51:     LOADFROMADDR
52:     PUSHLOCADDR     1
53:     LOADFROMADDR
54:     ADD
55:     SAVETOADDR
56:     PUSHLOCADDR     -2
57:     LOADFROMADDR
58:     PUSHNUM         10
59:     LT
60:     JUMPONFALSE     66
61:     PUSHSTATADDR    2
62:     PUSHLOCADDR     -2
63:     LOADFROMADDR
64:     SAVETOADDR
65:     JUMP            70
66:     PUSHSTATADDR    2
67:     PUSHLOCADDR     1
68:     LOADFROMADDR
69:     SAVETOADDR
70:     PUSHSTATADDR    2
```

`fixUpOperand()` is an important method of `OneOperandInstruction.java`. If `instr` is a reference to a one-operand VM instruction then `instr.fixUpOperand(k)` sets that instruction's operand to the integer k.

So, after generating the `JUMP a` instruction, `whileStmt()` can fix up the operand of the previously generated `JUMPONFALSE Instruction.OPERAND_NOT_YET_KNOWN` instruction as follows:

```
jInstr.fixUpOperand(Instruction.getNextCodeAddress());
```

Also Study the Method `ifStmt()` [lines 555 – 85]

The above hints should provide enough information for you to complete the method `whileStmt()`. But you should also study the method `ifStmt()`, which has already been written for you and provides further examples of how `Instruction.OPERAND_NOT_YET_KNOWN`, `Instruction.getNextCodeAddress()`, and `fixUpOperand()` are used. Here the relevant EBNF rule is

```
<ifStmt>  ::=  if '(' <expr7> ')'  <statement>  [else <statement>]
```

and the code to be generated is as follows:

Case 1:

```
<ifStmt> ::=  if '(' <expr7> ')' <statement>1

     <ifStmt>.code =      <expr7>.code
                          JUMPONFALSE a
                          <statement>1.code
                       a:
```

Case 2:

```
<ifStmt> ::=  if '(' <expr7> ')' <statement>1 else <statement>2

     <ifStmt>.code =      <expr7>.code
                          JUMPONFALSE a
                          <statement>1.code
                          JUMP b
                       a: <statement>2.code
                       b:
```

It is *possible* that there will be one or more questions on Exam 2 or the Final Exam which test your understanding of the method `ifStmt()`.

Hints Relating to the Gaps on Lines 492, 495, and 511 in ParserAndTranslator.java

Here the relevant EBNF rule is:

```
<assignmentOrInvoc> ::= IDENTIFIER ( { '['<expr3>']' } = <expr3> ; | <argumentList> ; )
```

This rule has two cases:

Case 1: `<assignmentOrInvoc> ::= IDENTIFIER { '['<expr3>']' } = <expr3> ;`

Case 2: `<assignmentOrInvoc> ::= IDENTIFIER <argumentList> ;`

The gaps you have to fill in relate only to Case 1. In that case the code to be generated is as follows:

```
<assignmentOrInvoc>.code = PUSHLOCADDR IDENTIFIER.stackframe_offset  or  PUSHSTATADDR IDENTIFIER.address
                           LOADFROMADDR
                           <expr3>index 1.code
                           ADDTOPTR
                                .
                                .
                           LOADFROMADDR
                           <expr3>index k.code
                           ADDTOPTR
                           <expr3>right side.code
                           SAVETOADDR
```

where `<expr3>`$_{right\ side}$ means the expression on the right side of `=`, and where the number of occurrences of

```
LOADFROMADDR
<expr3>index i.code
ADDTOPTR
```

is equal to the *number of indexes* after the `IDENTIFIER`—i.e., the number of times `'['<expr3>']'` occurs. Often there are no indexes (i.e., the assignment is of the form `IDENTIFIER = <expr3>`$_{right\ side}$ `;`). In any case the loop on lines 500–507 generates `LOADFROMADDR`, `<expr3>`$_{index\ i}$`.code`, and `ADDTOPTR` as many times as is needed. But you must fill in the ***gap on line 511*** in such a way that `<expr3>`$_{right\ side}$`.code` and `SAVETOADDR` are generated.

Note that the compiler must *generate* `PUSHLOCADDR IDENTIFIER.stackframe_offset` if the `IDENTIFIER` is a local variable or formal parameter, but it must instead generate `PUSHSTATADDR IDENTIFIER.address` if the `IDENTIFIER` is a static variable. To determine whether the `IDENTIFIER` is a local variable / formal parameter or a static variable, and to determine its stackframe offset in the former case and its data memory address in the latter case, the compiler looks up the `IDENTIFIER` in the ***symbol table***. The symbol table is a table, maintained by the compiler, which contains:

1. A `LocalVariableRec` object for each parameter and each local variable that is in scope at the point the compiler has reached in the program it is compiling.
2. A `ClassVariableRec` object for each static variable whose declaration has been seen by the compiler.

The compiler records information about each variable or parameter in its `LocalVariableRec` or its `ClassVariableRec` object. For example, it stores the data memory address of each static variable in the `offset` field of that variable's `ClassVariableRec` object. Similarly, the compiler stores the stackframe offset of each parameter or local variable v in the `offset` field of v's `LocalVariableRec` object. (The symbol table also contains a `MethodRec` object for each method that has been declared or called in the part of the program that has been seen by the compiler. But you will *not* have to write any code that deals with `MethodRec` objects to complete Assignment 2.)

Line 480 of ParserAndTranslator.java sets `identName` to the name of the `IDENTIFIER`. On line 485, `t = symTab.searchForVariable(identName);` looks in the symbol table for the `IDENTIFIER`'s `LocalVariableRec` or `ClassVariableRec` object, and sets `t` to refer to that object. Therefore the Boolean value of `t instanceof LocalVariableRec` on line 491 will be **`true`** or **`false`** according to whether the `IDENTIFIER` is a local variable / formal parameter or a static variable. In the former case `t.offset` will contain `IDENTIFIER.stackframe_offset`; in the latter case `t.offset` will contain `IDENTIFIER.address`. Use `t.offset` to fill in the ***gaps on lines 492 and 495*** in such a way that `PUSHLOCADDR IDENTIFIER.stackframe_offset` is generated if the `IDENTIFIER` is a local variable or formal parameter, but `PUSHSTATADDR IDENTIFIER.address` is generated if the `IDENTIFIER` is a static variable.

x: local var with stackfram offset +4
x = 234; //code to be generated

pushloc addr +4 — pushes ptr to x
pushnum 234 — pushes 234
save to addr — pops 234, pops ptr to x; saves 234 into x

if x were a static var with addr 3, ther
push stat addr 3
is needed instead of pushloc addr