CS 316 3/17/2014.

Another way to making an argument smaller for recursion:
(cdr e1) where e1 is a suitably transformed version of e.

Ex ssort on Asn. 5
(ssort L) ⇒ a list obtained by sorting the element of L into ascending order.
L: List of real numbers e
(ssort '(3 7 1 0 8 9 5)) ⇒ (0 1 3 5 7 8 9).

In this case, let e1 = (Min-first e).
= e with the first occurence of the ~~least~~ least element moved to the ~~first~~.

e1 = (0 3 7 1 8 9 5)
(cdr e1)

Sometimes you should use different recursive strategies for different argument values.

Ex Merge-lists. on Asn 5 (Asn = Assigned ϕ, Assignment).
(merge-lists L1 L2) ⇒ two lists of real members in ascending order
a List of members in ascending order obtained by merging L1 and L2

(merge-lists '(2 7 9 12) '(0 1 8 10 15)) ⇒ (0 1 2 7 8 9 10 12 15)
L1 L2

strategy 1: compute (merge-lists L1 L2) from (merge-lists (cdr L1) L2).
strategy 2: compute (merge-lists L1 L2) from (merge-lists L1 (cdr L2)).
strategy 1 is appropiate of (car L1) < (car L2)
strategy 2 is appropiate of (car L1) > (cad L2)
Either strategy works of (car L1) = (car L2)

Another Example unrepeated-elts (from Asn. 5)
(unrepeated-elts '(C A D D A B B B A C)) ⇒ (C A D A B C)
~~strategy~~ strategy 1: compute (~~unrepeated~~ unrepeated L) from (unrepeated elts (cdr L))
strategy 2: — — — (unrepeated (cddr L))

L = (CA DD . .) ~~target~~ strategy 1

L = 0(AADD . . .) strategy 2

L = (A A A — —) storge 3

Programming paradigm.

Already discussed:

Imperative programming

object-oriented imperative programming

Functional programming

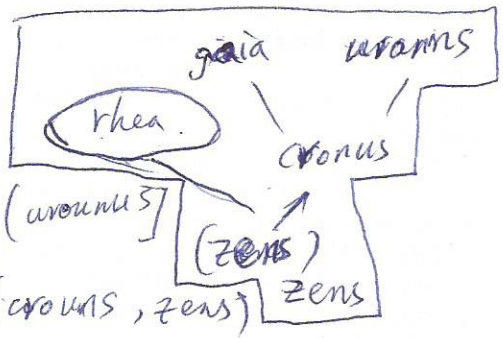
Aother Large. established paradigm in logic programming

In logic programming we write collecting of facts and inference rules.

A logic pragrm is used by writing queries which the logic programming system will find answers to by deduction from the facts and inference rules.

Prolog (developed in the early 1970s by colmerauser) is the best known logic programming Language.

Simple Example:



parent (uranus, cronus)

parent (cronus, zens)

Facts

parent (gaia, crouns)

parent (rhea, zeus)

inference rule.

grand parent (person1, person2)

:- parent (X, person2)

parent (person1, X).

the above is a very simple logic program (consisting of 4 facts clauses and 1 rule clause).

Question that use their program: ?- parent (gaia, X)

X = cronus; ⟶ "try to find another answ

false = no more answers.

?- parent(X, zeus)

X = rhea;

X = cronus;

false;

?- grandparent(Y, zeus);

Y = gaia;

Y = uranus;

false;

Syntax of programming Languages.

You will receive exercises soon — check email regularly (by Friday).

Syntax = form without regard to meaning

Semantics = meaning

syntax rules have the property that a compiler can check whether or not they are violated — it is never necessary to execute the program to determine whether a syntax rule is violated.

Analogy with English

The dog barked loudly.

|| A valid sentence. == syntactically valid has correct grammtic form & semantically valid ("meaningful")

The night barked loudly.

|| syntactically valid* — Has the same form/syntax as the above valid sentence

|| semantically invalid ("meaningless" — because. nights cannot bark).

* if can be obtained from that sentence by replacing one name with another.

The dog loudly. → syntically invalid — don't have correct grammatical form as it has no v

For a program in a language such as C++ or Java, a ~~syntical~~ syntactically valid program (roughly speaking), a piece of text that can be obtained from some valid program by making changes of the following kinds (zero or more times).

1. Remove a declaration (of a variable, function etc.)
2. Replace one identifier with another
3. Replace a constant in literal (e.g. "dog" or 37) with another literal of the same kind.

Ex: A syntactically valid C++ programm that is not a valid C++ progr

```
int g()
{
    int x=7;
    return x[3]/((x-7)/(3-3));
}
```

This is syntactically valid because it can be obtained from the following valid program by making changes of the above kinds:

Exercise

Remider

Do the reading assignment and exercises (on the syntax of expression) before Exam!

Lisp Double Recursion ("divide-and-conquer")

A doubly recursive function computes its result from two different recurise calls (for at least some argument values).

No. of recursive calls grows exponentially with the depth of recursion. (But in many doubly recursive function the depth of recursion grows only Loganthmically with argument size and so the no. of recursive calls does not grow exponentially with argument size).

Ex MSORT from Asn 5.

(msort L) ⇒ a sorted list of the elements of L. (L → List of real number, ascending order.)

(msort '(3 7 1 6 5 8 2))

⇒ (1 2 3 4 5 6 7 8).

(msort '(2 7 3 0 4 5 8))

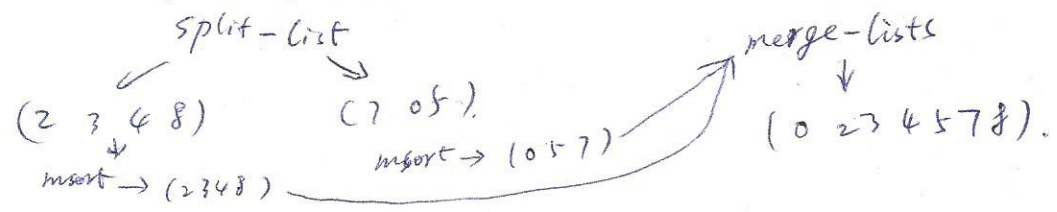
(msort '(2 3 4 8)) (msort '(7 0 5))

(2 3 4 8) (0 5 7)

(merge-list '(2 3 4 8) '(0 5 7))

(0 2 3 4 5 7 8).

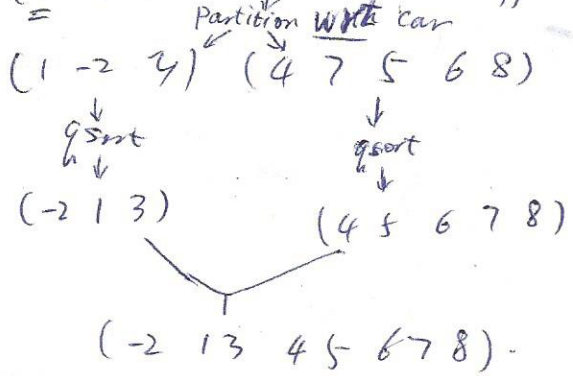
Aother picture (2 7 3 0 4 5 8).



Another example

(Qsort L) ⟹ same result as (Msort L).

(Qsort '(4 1 7 -2 5 6 8 3)). wrt = with respect to

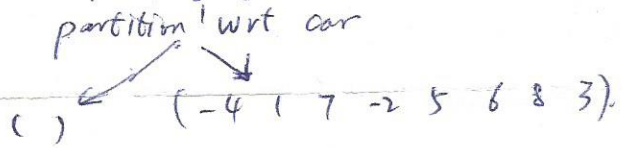


When does the above recursive strategy fail?

① L is NIL

or ② First element of L is the first element is the smallest element of the list.

L = (-4 1 7 -2 5 6 8 3).



A special case of ② is when L is of length 1, but this could be handled as a base case. A different recursive strategy is needed in case ②!

Functions that take functions as arguments

Ex: $\sum_{i=2}^{4} i^2 = 2^2 + 3^2 + 4^2 = 29$

(defun sqr (n) (* n n))

Ex: (sigma #'sqr 2 4) ⇒ 29

How can we write function SIGMA that is like Σ?

In Common Lisp [this does NOT apply to scheme] the value of a symbol as a variable (if any) is not related to the function definition of that symbol (if any)

SETF is one way to set the value

LET/LET* is another way to set the value.

Most importantly when a function is called, the value of each formal parameter is set to the value of the corresponding actual argument.

DEFUN sets the function definition of a symbol. (DEFUN has no effect on the value of the symbol.)

```
(defun sqr (n) (* n n))
(setf sqr 5)
```

```
> (sqr sqr)
25
```

#'f is an expression whose value is the ~~defini~~ function definition of f.

This explain why we wrote (sigma #'sqr 2 5) and NOT (sigma sqr 2 5).

~~Function~~ Funcall

(funcall g a_1 a_2 a_3 ... a_k).

calls the function given by the value of g with a_1, ..., a_k as the actual arguments.

```
(setf g #'sqr)
(defun g (n) (+ n 17))
>(g 3)
20
```

```
> (funcall g 3)   ⇐ g = sqr.
9
```

```
> (funcall #'g 3)
20
```

In scheme, there is no difference between function definition and value. So FUNCALL and #' are not used in scheme.

Now we are ready to write the SIGMA function.

```
(defun sigma (f m n)
  (if (> m n)
      0
      (+ (funcall f m)
         (sigma f (+ m 1) n))))
```

$$\sum_{i=m}^{n} f(i) = f(m) + \cdots + f(n)$$

$$= \begin{cases} f(m) + \sum_{i=m+1}^{n} f(i) & \text{if } m \le n \\ 0 & \text{if } m > n \end{cases}$$

CS316 3/19/2014 ⑥

MAPCAR (map in scheme)

(mapcar #'round '(7.1 6.8 3.2 3.1 5.9))

⇒ (7 7 3 3 6)

(mapcar f L) ⇒ a list of same length as L in which the ith element is obtained by appling f to the i^{th} element of L.

(mapcar #'eqr (2 6 3 9)) ⇒ (4 1 36 9 8)

Let's write our own version of Mapcar.

(defun our-mapcar (f L)

L = e_1 e_2 ... e_k) our mapcar (f L) should ⇒ (f(e_1) f(e_2) ... f(e_k))

(defun our-mapcar (f L)

(if endp L)

()

(cons (funcall f (car L))

(our-mapcar f (cdr L)))))

The build-in mapcar Mapcar (unlike our-MAPCAR) can also be used to map function of 2 or more arguments.

(mapcar #'+ '(7 6 3) '(2 4 5 8) '(0 1 6 1)) ⇒ (3 12 17 12)

(mapcar #'cons '(A B C) '((1 2) (3 4 5) (6)))

⇒ ((A 1 2) (B 3 4 5) (C 6))

Lambda Expression (also available in Java, and any other res 8)

Lambda expression evaluate to nameless function.

Syntax is the same as for DEFUN but without the function name.

(lambda (n) (* n n))

((lambda (n) (* n n)) 5) ⇒ 25

(sigma (lambda (k) (* k k)) 2 4) ⇒ 29. $\sum_{k=2}^{4} k^2$

Lamda Expression (contd.)

> ((lamda (x y z) (+ x (* y z))) 2 1 7)

=

> (mapcar (lambda (x) (+ x (* x x x))) '(2 1 0 4))

(10 2 0 64)

> (mapcar (lambda (x y) (+ x (* y 2))) '(1 0 2 3) '(0 1 1 2)) ⇒ (1 2 4 7)

Σ (sigma (lambda (i) (+ (* i i) (* i 3))) 0 3) $\sum_{i=0}^{3} (i^2 + 3i)$.

$\Rightarrow 0^2+3*0 + 1^2+3*1 + 2^2+3*2 + 3^2+3*3$

$\Rightarrow 32$

Important A lambda expression can use variables defined outside that lambda expression relevant to FOO problem (Q14 on Asn 5)

Ex: Write a function SUM-POWERS such that

(sum-powers k m n) $= \sum_{i=m}^{n} i^k = m^k + (m+1)^k + \cdots + n^k$.

```
(defun sum-powers (k m n)
    (sigma (lamda (i) (expt i k)) m n))
```

Ex: INC-LIST-2: from assignment.

(inc-list-2 '(2 7 1 65) 3) ⇒ (5 10 4 9 8)

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

Apply Apply is like funcall but allows a list of arguments to be passed.
*last argument must be a list.

(apply #'+ '(2 1 7 9))
= (funcall #'+ 2 1 7 9).
= (+ 2 1 7 9) = 19.
= (apply #'+ 2 '(1 7 9))
= (apply #'+ 2 1 '(7 9))
= (apply #'+ 2 1 7 '(9))
= (apply #'+ 2 1 7 9 ())

(apply f i_1 i_2 ... i_k L) ← a list
calls f with the values of i_1 i_2 ... i_k and the elements of the value of L as arguments.

(funcall f i_1 i_2 ... i_k)
= (apply f (list i_1 i_2 ... i_k))
= (apply f i_1 (list i_2 ... i_k)) ... etc.

Ex. Sum from Asn-4
[(sum '(2 7 1 6)) ⇒ 16]

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

(apply #'cons '((+ 2 3) (+ 4 A))) ⇒ ((+ 2 3) + 4 A)
↗ this is not + here.

Ex: (relevant to 16c).

```
(apply #'mapcar #'+ '((1 2 3)
                      (4 0 7)
                      (0 1 1)
                      (3 3 2)))
```

= (mapcar #'+ '(1 2 3) '(4 0 7) '(0 1 1) '(3 3 2)) ⇒ (8 6 13)

Syntax

Syntax of Expression.

Expression can be written in many different notation / syntaxes.

Consider the expression : $f(g(h(1,2), f(3,4)), 5)$ (in java/C++ syntax)

This expression can be written in.

① infix notation as : $((1\ h\ 2)\ g\ (3\ f\ 4))\ f\ 5$
$((1 * 2) - (3+4)) + 5$

② infix notation, with f and g having equal precedence. and belonging to a left associative precedence class, h having higher precedence than f and g.

$1\ h\ 2\ g\ (3\ f\ 4)\ f\ 5$. (if $f=+$, $g=-$, $h=*$)

$1 * 2 - (3 + 4) + 5$

③ 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 assumes you know how many arguments each operator/function takens]

⑤ "anti-Lisp" notation — like Lisp but function names appear at the ends of lists (instead of ~~that~~ at the begining s of list).

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

⑥ Postfix notation = anti-Lisp without parentheses.

$1\ 2\ h\ 3\ 4\ f\ g\ 5\ f$.

Syntax of infix notation

Terminology operator = symbol that denotes a function ~~the~~

The arity of an operator is the no. of arguments taken by the function

Each argument maybe called an operand.

An operator of arity n is called an n-ary operator

binary = 2-ary
unary = 1-ary
ternary = 3-ary.

CS316 3/24/2014 ⑥

Syntactically valid infix expressions (SVIEs) can be defined as follows:

e is a SVIE just of one of the following is true:

① e is a constant or a variable

② e is (e_1) where e_1 is an SVIE.

③ e is e_1 op e_2 where each of e_1 and e_2 is an SVIE and op is a binary operator

④ e is op e_1 where e_1 is an SVIE and op is a prefix unary operator

⑤ e is e_1 op where e_1 is an SVIE and op is a postfix unary operator.

prefix unary operators

$-y$
$-s$
$--x$
$*s$

x++ postfix unary o

This definition of SVIE's is a serious weakness. Some syntax decompositions of SVIEs that are suggested by this definition are inconsistent with the desired semantics.

Ex: $e = x + y * z - w$

Can be decomposed using value ③ into

e_1	*	e_2
//		//
$x+y$		$z-w$

This is inconsistent with the usual semantics of arithmetic expressions.

After the exam, you will study sec 2.5 which gives another way to specify SVIES that is unambiguous specification that is consistent with given precedence and associativity rules.

One way to fix the problem is to add to ③, ④, and ⑤, the rule that op should be the operator that is applied last, and to give rules that determine which operator is applied last.

CS 316. Read assignment (P49) Sethi

2.1 a) $a*b+c$ = * + a b c

b) $a*(b+c)$ = + * b c a.

c) $a*b+c*d$ = + * a b * b d.

d) $a*(b+c)*d$ = * * a + b c d.

e) $(b/2 + sqrt((b/2)*(b/2) - a*c))/a$ =

= / - + / b 2 sqrt * / b 2 / b 2 * a c a

2.2. a) $a*b+c$ = a b * c +

b) $a*(b+c)$ = a b c + a *.

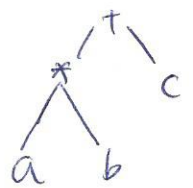
c) $a*b+c*d$ = a b * c d * +.

d) $a*(b+c)*d$ = b c + a * d *

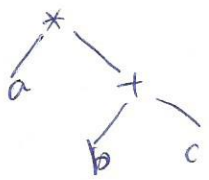
e) $(b/2 + sqrt((b/2)*(b/2) - a*c))/a$

= b 2 / b 2 / b 2 / * a c * - sqrt + a /

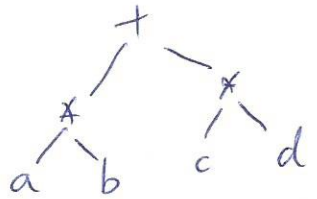
2.3. a)



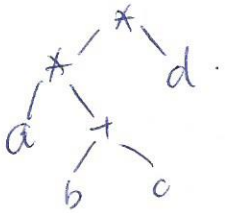
b)



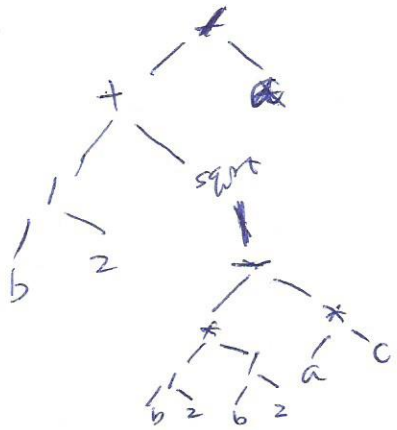
c)



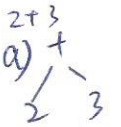
d)



e)



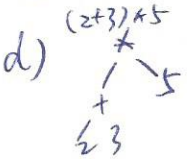
2.7. a)



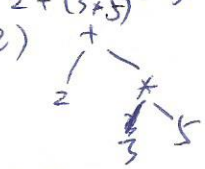
b)

c)

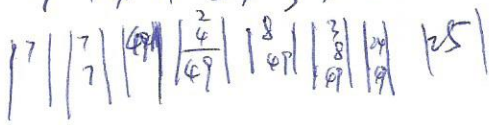
d)



e)



2.8. 7 7 * 4 2 * 3 * -



CS316 3/26/2014 ①

e is e_1 op e_2

SVIE SVIE

a+b*c

op op

decomposition a+b*c

e_1 op e_2 ← bad decomposition ✗

a+b * c

a+b * c ← good decomposition. ✓

e_1 op e_2

a + (b*c) / d

↑ op ↑ op

How precedence and associativity Rules of an infix Notation are used to determine which operator is Applied Last.

Warning Precedence and associativity rules don't always uniquely determine which operator is applied first.

Ex: y / z * --y

applied last

Assuming -- belongs to a higher precedence class than * and / and that * and / belong to the same precedence class which is a left associative class.

The precedence and associativity rules imply * is applied last, but do not say which operator is applied first.

Some languages may have additional rules that do determine which operator is applied first. Ex: Java has a left to right evaluation rule, which implies that / is applied first in the above expression. [C++, on the other hand, does not have this rule in C++ it is upto the compiler whether -- or / is applied first in the above expr.

```
int y=4
System.out.println( y/z* --y )  prints 6 in java
                                [would print 3 of -- were applied first).
```

Operators are partitioned into ranked precedence classes.

Each precedence class is specified as being left-associative or right-associative.

To determine which operator in an expression e is applied first:

① Let e' = e without its ~~same~~ surrounding parentheses if any.

Ex. e = (x + (3*(y-7/w)) * u)

e' = x + (3*(y-7/w)) * u.

↑ ↑

top level means not in parentheses.

② Find all the top-level operators in e' [top-level means "not surrounded by parentheses"]

③ Among the operator is found at top ②, find the operators of lowest precedence.

④ if just one operator is found at step ③, then that operator is the operator that is applied last. Otherwise, the operator that is applied last is the left most (right most) of

⑤ the operators found at step ③ / according to whether their precedence class is ~~right~~ right- (left-) associative.

	prefix unary	postfix unary	binary	associativity
Highest ↑ class 1	~			right
class 2	+ -		+ -	left
class 3			& ^ @	right
class 4			# $	left
↓ Lowest				

which operator is applied last in

① - x + z $ (~ y ^ z) & a @ ~ z ^ x # y - 1 (left associativity)

② + x @ (z & ~ y ^ z) + (a @ ~ z ^ x) & y - 1. (right associativity)

Postfix and prefix Notations

prefix notation = Lisp notation without parentheses.

postfix notation* = "anti-lisp" notation without parentheses.

* also called reverse Polish notation or RPN.

In these notation you need to know the arity of each operator. You cannot use the same symbol to denote operators of different arities.

-- x y could mean (- (- x) y) or (- (- x y))

(-$_2$ (-$_1$ x) y) (-$_1$ (-$_2$ x y))

-$_2$ -$_1$ x y (-$_1$ -$_2$ x y)

We should use different symbols for binary and unary - to avoid this problem.

Differences between Postfix/Prefix Notations and infix Notation

① Operator may have any arity in postfix/prefix notation [they may only have arity 1 or 2 in infix notation]

② No parentheses in postfix/prefix notation

③ No precedence classes are used with postfix/prefix notation.

④ In postfix/prefix notation the same symbol should not be used to denote 2 operators of different arities.

Syntactically valid prefix/postfix expression can be defined as follows:

① A constant (5) or a variable (w) is a syntactically valid postfix/prefix expression

② If op is an operator of arity k, then.

(a) op $e_1 \cdots e_k$ is a syntactically valid prefix expr if each of the e's is such an expr.

(b) $e_1 \cdots e_k$ op is a syntactically valid postfix if each of the e's is such an expr.

Semantics In case ②, the expr. is evaluated by evaluating each of the e's and applying op to the values of the e's

Postfix expression can be evaluated using a stack; ~~Reading the expr. from left to right.~~

- Reading the expr from left to right
- whenever you see a constant or variable, push its value
- whenever you see an operator of arity k, pop k values, apply the operator to those k value and push the result.

When the entire expr has been processed, its value will be the only thing on the stack.

~~i^{th}~~ → i^{th} value to be popped is the ~~k^{th}~~ i^{th}-last operand.

Prefix expression can be evaluated in the same way, ~~expe~~ except that the expression is read from right to left and when k operands are popped the i^{th} operand to be ~~popp~~ popped is the ith operand of op (not the i^{th}-last)

You can use essentially the same algorithms to "put the parentheses back in a postfix /prefix expr. (to produce anti-lisp/lisp notation).

CS716 3/26/2014 (6)

Lisp $(+_3\ (-_2\ (+_4\ x\ y\ (*_2\ 5\ x)\ w)\ u)\ v\ (*_3\ x\ y\ 7))$

prefix. $+_3\ -_2\ +_4\ x\ y\ *_2\ 5\ x\ w\ u\ v\ *_3\ x\ y\ 7$

TinyJ Assignment 1 email*: Do the installation before next Monday.
* with assigned reading and exercises on syntax and method overriding in C++.

Syntax of Programming Languages.

The syntax of programming language is usually specified in terms of

① Units called tokens, such as: while ①, for ②, ~~IDENTIFY~~ IDENTIFIER ③ (x, while, for)
UNSIGNED-INT-LITERAL ④ (72. 47193) ; ⑥ (⑦ >= ⑤ - ⑧ (java/C++ tokens).

some tokens (e.g. IDENTIFIER and UNSIGNED-INT-LITERAL in Java/C++) have more than one instance.

Different instances of the same token are semantically different (i.e. have different meanings) but syntax specification written in BNF or some similar notation such as EBNF will not distinguish between them.

Note: It is also possible for the same instance of token to have more than one spelling.

Apple 73.0
AppLE 73.00

A token (like while or for in java/C++) that looks like an identifier but cannot be used as identifiers are called reserved words. [Some languages, such as Fortran and Lisp, has no reserved words, in such languages words that have special meaning (like IF in Lisp) are called keywords Some author refer to ~~reserved~~ reversed words as keywords.

[(defun nil (if (+ if 1)) is a legal lisp function definition]

② Certain choosen language constructs (e.g. <while stmt>, <stmt> or <expression>). In a syntax specification of an entired language (rather just some part of a language) one of these constructs represents acceptable input for a complier or interpreter — e.g. it might be called <program>.

The purpose of a syntax specification is to answer the following 2 questions:

Q1: For each kind of token exactly which sequences of characters constitute a valid instance of that token.

Q2 For each of the language constructs of ②, exactly which sequences of tokens constitute a syntactically valid instance of that language construct?

[Reminder] A syntactically valid (x = 42*(y-w)) instance of a construct need not be a valid instance — it may contain errors such as undeclared identifier type mismatch or devided by zero.

Q1 and Q2 are usually answered using context-free grammar notation / BNF or some variant of this (such as EBNF)

Context-Free grammar Notation is a relatively concise (and yet completely price) notation for specifying a (possible infinite) set of finite sequences of symbols. The symbols are called the Terminal of the grammer.

In answering Q1 (for tokens that have many instances, like IDENTIFIER or Unsigned-float-Litera) we can use a grammer whose terminals are character; [Regular expression notation is another way to do this]

In answering Q2, we can use a grammer whose terminal are tokens

It is also possible to use a single grammar, whose terminals are character, to answer both Q1 and Q2.

Context-free grammer notation was intended by Chamsky, who invented a hierarchy of 4 types of grammer for specifying the syntax of English and other similar natural languages. Context-free grammars are Type 2 in Chamsky's Hierarchy.

the idea of a context-free grammar was independently reinvented by Backus a few years later. Backus proposed the use of his grammar notation for specifying the syntax of Algol 60.

the Algo 60 Report (which specified the Language Algol 60 in a way that was widly admired by computer scientists) adopted Backus's proposal.

Naur was the editor of the Algol 60 Report. Backus's notation is now call BNF (Backus Naur Form)

<E> ::= <E> + <T> (BNF)

E → E + T. (not BNF)

Like many authors today (A) we will use BNF to mean some ~~of com~~ commonly used notation for writing a context-free grammer.

(A) But unlike Sethi:

cf Fig 2.6 on ——P42.
and Fig 2.10 on P46.
Sethi would not consider Fig 2.6 to be BNF.
But we will consider it to be BNF.

Ex of a Grammar.

UNSIGNED-FLOAT-LITERAL tokens

can be specified in BNF as follows: (assuming they have the form: ~~one or more digits. on~~ one or more digits . one or more digits. (Fig 2.3 on P36.)
ip (integer part) f (fraction) ← fraction.

① ufpl ::= ip . f

② ip ::= d ip ::= d | ip d. ("d" = "digit").

③ ip ::= ip d

④ f ::= d | d f

d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (15)

undefined constant symbols

The 11 symbols 0, 1, ., 9 are the terminals of this grammar.

The 4 symbols ufpl, ip, f, d are the nonterminals of this grammar.

↓
variables, each of which denotes of finite sequences of terminals.

There are 15 rules called productions ~~each~~ each of the form

N ::= α

N: a single nonterminal; α: a sequence of zero or more terminals and/or nonterminals.

The nonterminal ufpl in the starting nonterminal
This nonterminal denotes the set of sequences that the grammer is intended to define — this set is called the language of / generated by the grammar. Unless otherwise indicated, the starting nonterminal ~~on the left~~ is the nonterminal on the left side of the first production.

CS316 ① 4/7/2016.

(Final exam score) * 25/60

replace the lower of your scores on exam 1 and 2. if it is higher.

parse Trees

Last time it was mentioned that. Each nonterminal of a grammar represents a set of finite sequences of terminals.

grammer = context-free grammer = BNF specification. (in this course)

The set of sequences represented by the starting nonterminal is called the language of / generated by the grammar. [This is regarded as the set of sequences that is defined by the grammer]

① utpl ::= ip · f
2nd. 1st 3rd.

② ip ::= ②d / ip ②d. "1" means "the left side is the same as in the previous production"

f ::= ④d / df ⑤

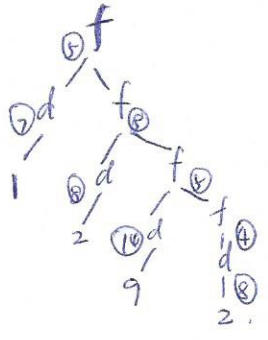
d ::= ⑥0 / ⑦1 / ⑧2 / ⑨3 / ... / ⑮9

the set of sequence of terminals that is represented by a nonterminal N is the set of all sequence of terminals that have parse trees rooted ~~at~~ at N.

important

If the root of a parse tree is not specified. then it should be assumed that the root is the starting nonterminal.

Here is a parse tree that proves: 1 2 9 2 ∈ f



A parse tree rooted at a nonterminal N is an ordered rooted tree with the following properties:

1. The root is N
2. each leaf is either a terminal or the symbol <empty>; <empty> leaf has no siblings.
3. each ~~internal~~ node (i.e. each internal node) is a nonterminal. non-leaf.

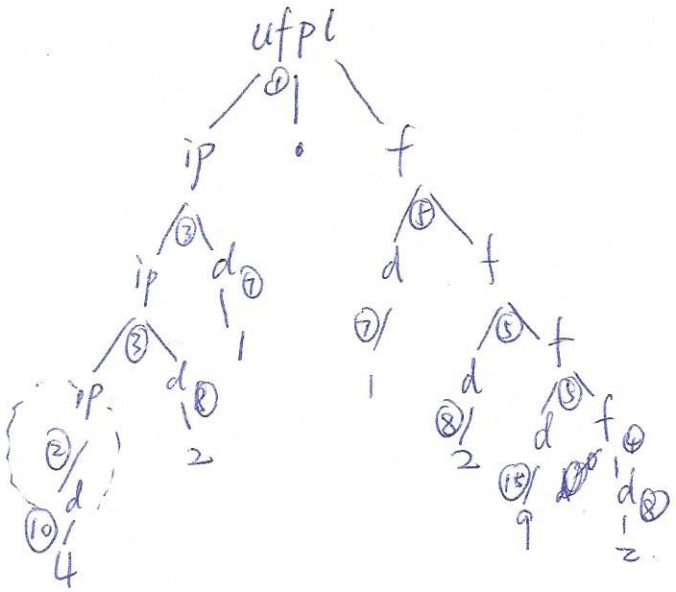
4. the sequence of children of each nonleaf node is the right side of a production whose left side is that node.

A parse tree whose sequence of non-<empty> leaves is $t_1\ t_2 \cdots t_k$ is called a parse tree of or a parse tree that generates $t_1, t_2 \cdots t_k$.

Here is a parse tree that parses

$$4\,2\,1.\,1\,2\,9\,2 \in ufpl$$

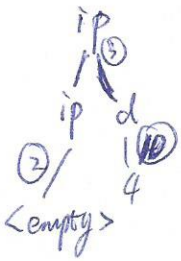
[ie. $4\,2\,1.\,1\,2\,9\,2$ ∈ the language of the above grammar].



Suppose we change production ② to ip ::= <empty>

Q. Is the above tree still a correct parse tree? No

A correct parse tree of $4\,2\,1.\,1\,2\,9\,2$ should have the following subtree instead of



CS316 ③ 9/7/2016

Extended BNF (EBNF)

EBNF = BNF + (...)$^+$ + [...]$^+$ + {...}

(...)$^+$: that mean parentheses

[...]$^+$: brackets, and braces

{...} : "0 or more times"

[parentheses in BNF has no special meaning; they are just ordinary characters]

[...] means "is optional";

Ex: e ::= (+|-)t (+|-)t. is a EBNF rule that is equivalent to the following 4 BNF productions

e ::= +t+t | -t+t | +t-t | -t-t

e ::= [+|-]t[(+|-)t] is equivalent to 9 BNF productions: [α] = α | <empty>

e ::= t | t+t | t-t | +t | +t+t | +t-t | -t | -t+t | -t-t

e ::= [+|-]t{(+|-)t} is equivalent to infinitely many production including:

e ::= t+t+t-t-t-t+t.

e ::= -t-t+t+t+t+t-t-t

e ::= t.

EBNF specifications can always be rewritten as a finite set of BNF productions.

One way to do this is to replace each occurrence of (...), [...], and {...} with a new nonterminal, which you must define.

(...) ⇒ N where N ::= ...

[...] ⇒ N where N ::= .... | <empty>

{...} ⇒ N where N ::= <empty> | N...

{α|β|γ} ⇒ <empty> | N(α|β|γ) ⇒ N ::= <Empty> | Nα | Nβ | Nγ.

Work from the inside outwards

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

[+|-] : opt-sign, {(+|-)t} : op

⇒ e ::= opt-sign t rest

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

op ::= + | -

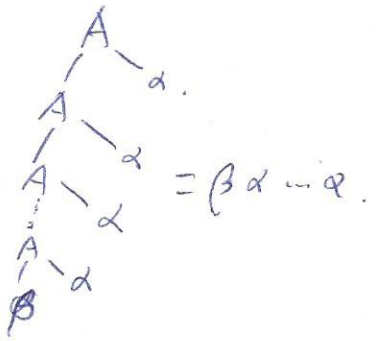
rest ::= <empty> | rest op t.

CS316 ④ 4/7/2016.

Useful Equivalences

$A ::= A\alpha \mid \beta \equiv A ::= \beta\{\alpha\}$

$A ::= \alpha A \mid \beta \equiv A ::= \{\alpha\}\beta$



$= \beta\alpha\alpha\alpha$.

Ex: A seq of one or more <stmt>s in which consecutive <stmt>s are separated by a semicolon.

EBNF: $\langle stmt\text{-}seq \rangle ::= \langle stmt \rangle \{ ; \langle stmt \rangle \}$ (~~BNF:~~) BNF: $A ::= A\alpha \mid \beta$.

$\langle stmt\text{-}seq \rangle ::= \langle stmt\text{-}seq \rangle ; \{stmt\} \mid \langle stmt \rangle$

seq of one or more d's

$ip ::= d\{d\} \Rightarrow ip ::= ip\,d \mid d$.

~~$f ::= \{d\}d \Rightarrow f ::= df$~~

$f ::= \{d\}d \Rightarrow f ::= df \mid d$.

$e ::= [+|-]+\{(+|-)+\}$

$\Rightarrow e ::= e\,(+|-)\,t \mid [+|-]\,t$

$\Rightarrow e ::= e+t \mid e-t \mid +t \mid -t \mid t$

euclid.cs.cuny.edu/3161

Symbols.java

Each enum constant in symbols.java ~~defines~~ represents a

→ terminal / token.

→ or a non-terminal or a "fictitious token"

```
method vardecl().
 TJ.outputprintSymbol(NTvardDecl)
        incTreeDept().
   if (getCurrentToken() == NT) {
        nextToken();
        SingleVarDecl();
        while (getcurrentToken() == comma) {
              nextToken();
              singleVarDecl();
        }
        accept(SEMICOLON);
  }
  else if(get currentToken() == Scanner) {
        ...
  }
```

int x; y=3; w;

<VarDecl>

reversed: int

<SingleVarDecl> , <SingleVarDecl> , <SingleVarDecl> ;

CS316 4/23/2014.

accept () → { accept current token
nextToken().

nextToken()

There is a case for calling accept(...) even when nextToken() would be adequate:
accept(...) would do a redundant check in such cases, but the advantage is that changes to the code that are implemented later would be less likely to bread the code.

TinyJ Virtual Machine (VM)

The memory of the TinyJ VM consists of 2 disjoint regions with separate addresses spaces:

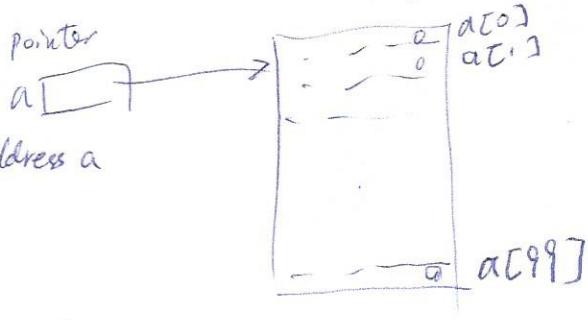
① Code memory
② Data memory

Code memory is used to store instructions that can be fetched for execution

[For assignment 2, there will be no fetching of instructions for execution, your program will just compile the TinyJ source file into a sequence of TinyJ VM instructions]

Data memory is used to store:
- values of variables.
- contents of arrays.
- characters of string literals

int[] a = new int[100];
a: variable. new int[100]: array



Data memory address a is unrelated to code memory address a

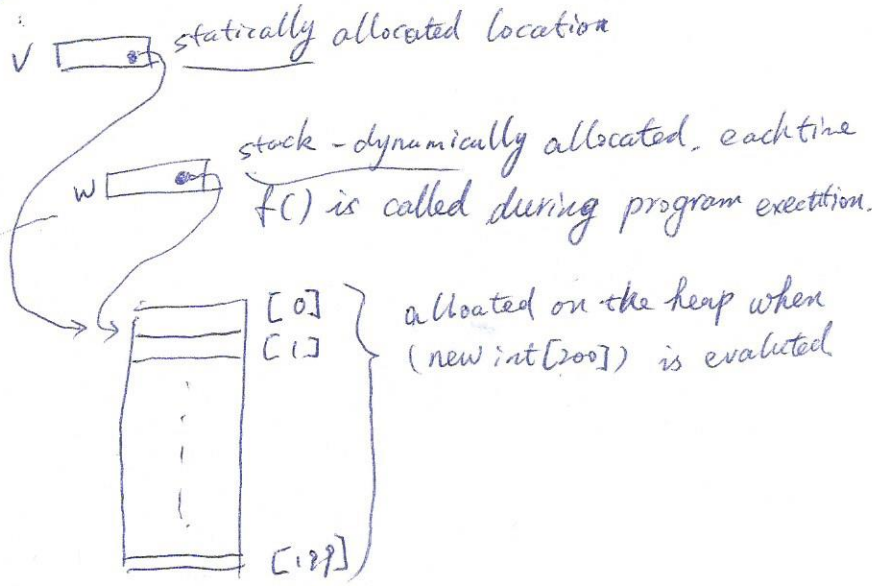
Data memory consists of 3 regions:

a) A statically allocated regions (used to store values of static variables and characters of string literals)
* In TinyJ, (unlike Java) strings are not objects, Arrays are the only objects in TinyJ

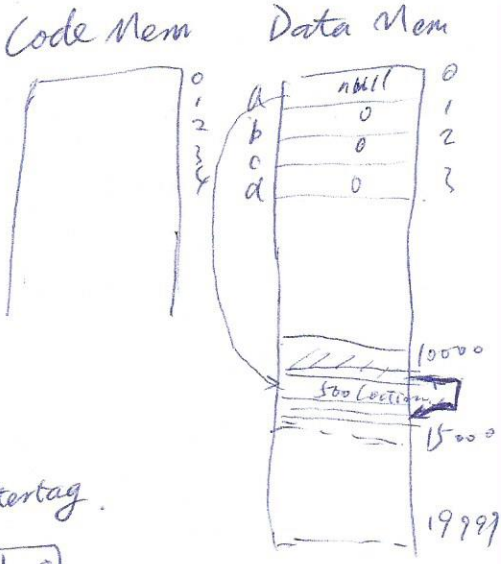
b) a stack-dynamically allocated region, used to store values of local variables of methods (including ~~formal~~ formal parameters of methods)

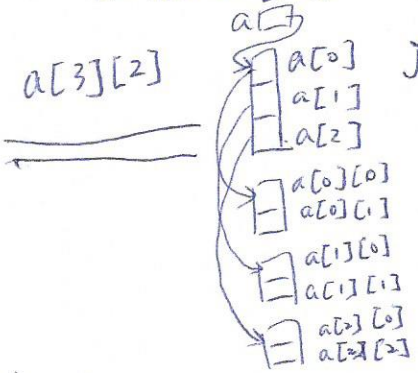
c) A heap-dynamically allocated region, used to store arrays (which are the objects in Tiny-J)

```
static int[] v[];
    :
void f () {
    int w[];
    w = new int[200];
    v = w;
```



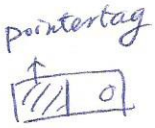
```
class prog {
    static int a[], b;
    static int c;
    static int d;
    a = new int [100]
```





Java:

```
int a[][] = new int [3][2];
```

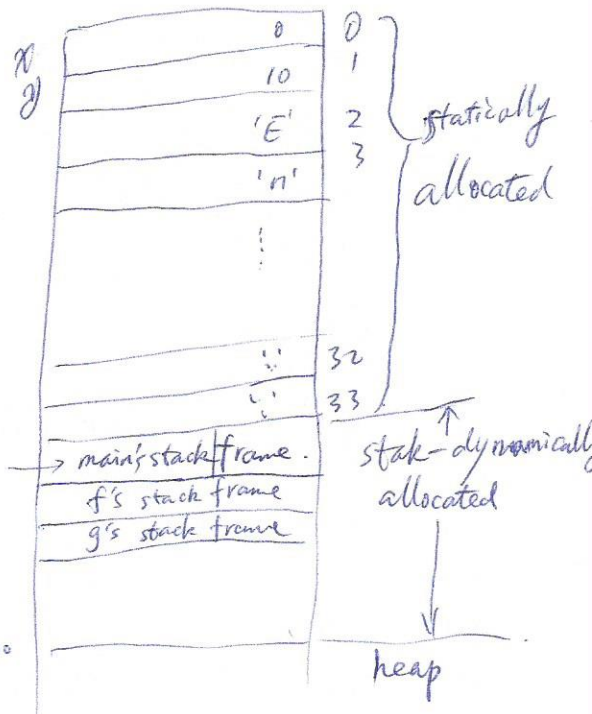



```
class {
    ...
    static int x, y = 10
    ...
    system.out.println ("345678910... ~");
    ...
```

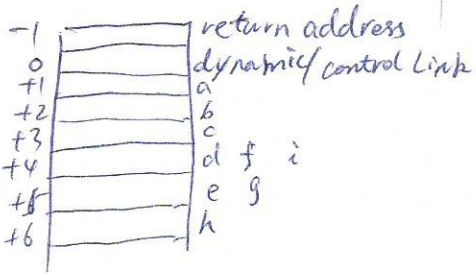
Address values.

Address values

example code:

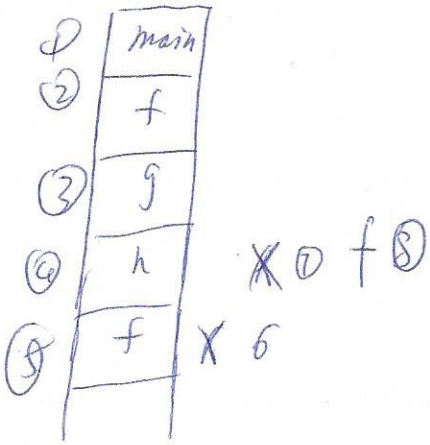



```
int func()
{   +1  +2  +3
    int a, b[], c;
    ....
    if (...)
        +4  +5
        int d, e;
        ....
    }
    else   4  5
    { int f, g;
      ...
      int h;
    }      4 ← ④
    int i;
}...
```



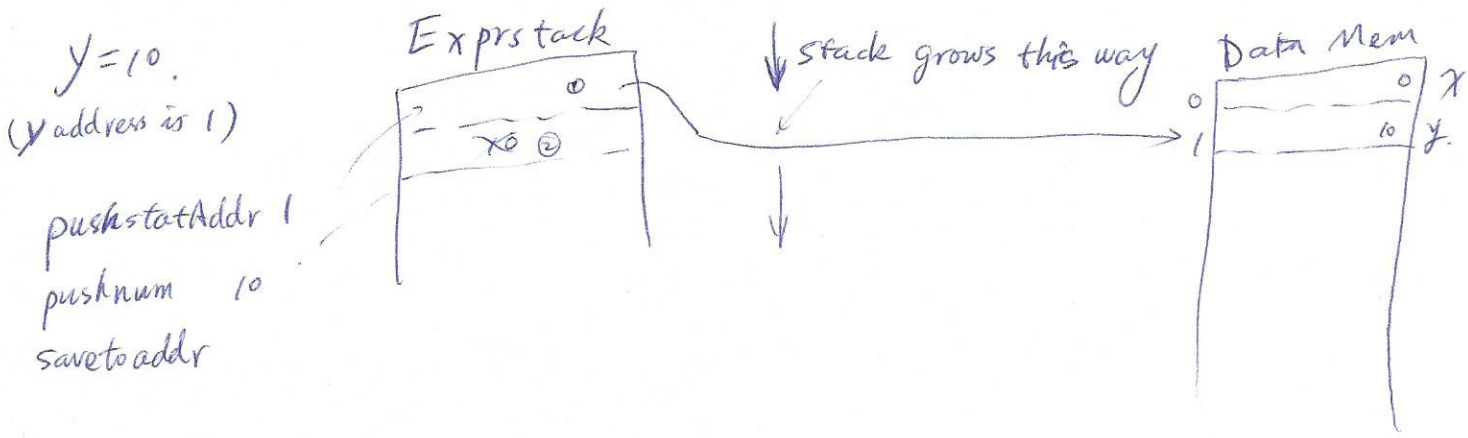
Cs516 4/28/2014 ②

VM - instruction-set-and-hints-for-asn-2.pdf.



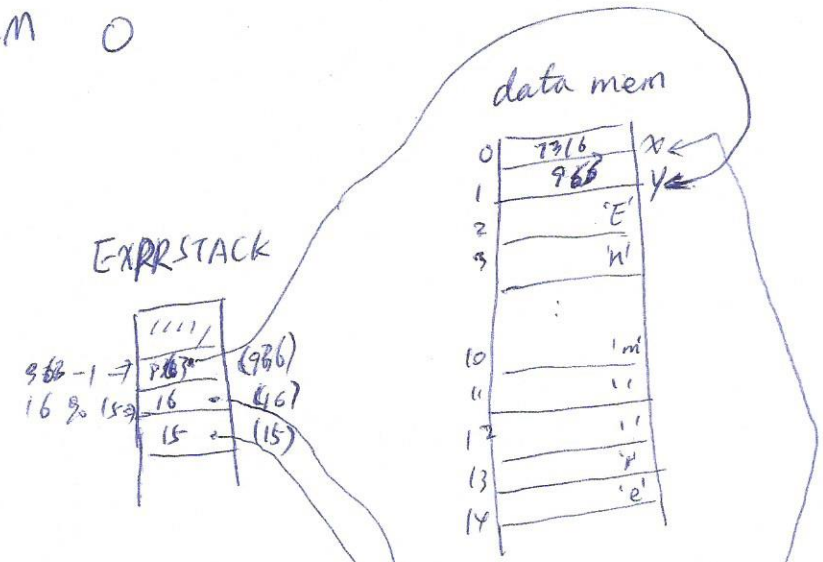
~~Exprstack is~~

Exprstack is the TinyJ VM's Expression Evaluation Stack.



main() is translated to: INITSTKFRM 0

```
INITSTKFRM   0
WRITESTRING  2   12
PUSHSTATADDR 0
READINT
SAVETOADDR
```

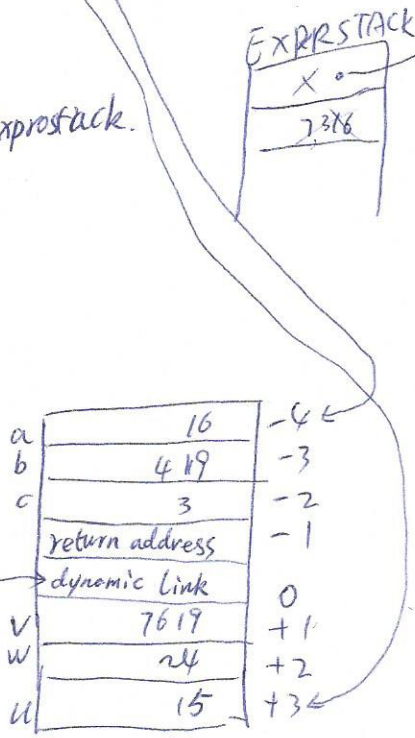


evaluate expression: y - a % u

Translation of return y - a % u; in method f()

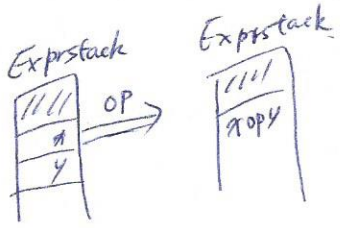
{ instruction that leave value of y - a % u on top of Exprstack.
return 3.

stack frame of each call of f() is as on follows.



To leave value of y - a % u on top of Exprstack.

```
f1: PUSHSTATADDR 1.
    LOADFROMADDR
    PUSHLOCADDR -4
    LOADFROMADDR
    PUSHLOCADDR +3
    LOADFROMADDR
    MOD
f8: SUB
```



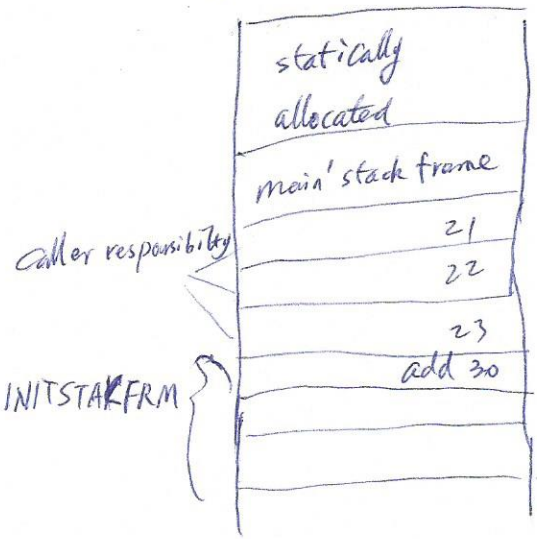
OP Maybe: ADD, SUB, DIV, MUL
MOD,
AND, OR
EQ, NE, LE, LT, GE, GT

Translation of f(21,22,23)

```
PUSHNUM 21
PASSPARAM
PUSHNUM 22
PASSPARAM
PUSHNUM 23
PASSPARAM
CALLSTATMETHOD 34.
```

in main()

Data MEM



Translation of

y = f(21, 22, 23).

```
PUSHSTATADDR 1
LOADFROMADDR
Code to call f(21, 23.23)
ADD
```

In Assignment 1, call a parsing method n() will read an <n> and output a parse tree with root <n> for that instance of <n>

In Assignment 2, that call of n() will also generate <n>.code for that instance of <n>.

Translation of

f(17, y, (x-y));

```
push NUM 17
passPARAM
y { pushSTATADDR 1
    LODASTATADDR
    PASSPARAM
x-y { PUSHSTATADDR 0
      LODASTATADDR
      PUSHSTATADDR 1
      LODASTATADDR
      SUB
      PASSPARAM.
```

CallSTATMETHOD 34

DISCARDVALUE.

Notation:

For any node <n> in the TinyJ same file parse tree, let

<n> code

mean the VM instruction generated by the compiler for <n>.

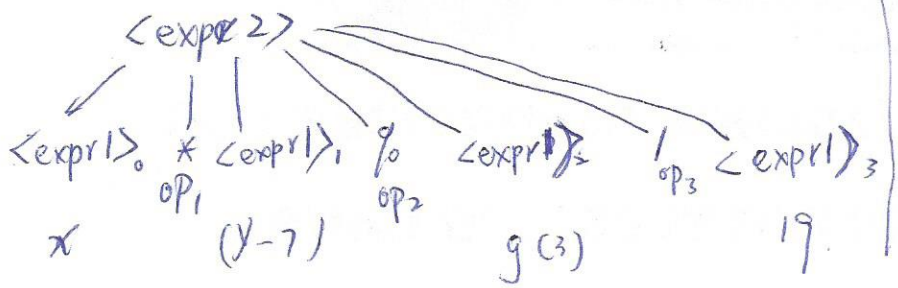
Ex: if <n> is the <statement> node for

f(17, y, x-y)

in the above program, then <n>.code = instructions 8-20 in the generated code

Example of an <expr2>:

x + (y - 7) % g(3) / 19



Assume: variable
x is static with address 3
Variable y is local with stack frame offset +4
g is a method whose code starts at address 29.

In this ~~exp~~ example

<expr2>.code = <expr1>0.code
PUSHSTATADDR 3.
LOADFROMADDR

<expr1>2.code
~~PUSH~~ PUSHSTATADDR 4
LOADFROMADDR.
PUSHNUM 7
SUB

~~<expr~~ OP1.instr
MUL

<expr1>2.code
PUSHNUM 3
PASSPARAM.
CALLSTATMETHOD 29

OP2.instr
MOD

<expr1>3.code.
PUSHNUM 19

OP3.instr
~~PA~~ DIV

new ADDinstr() generates ADD.

~~new PUSHNUMINST~~

new PUSHNUMinstr(23) generates PUSHNUM 23.

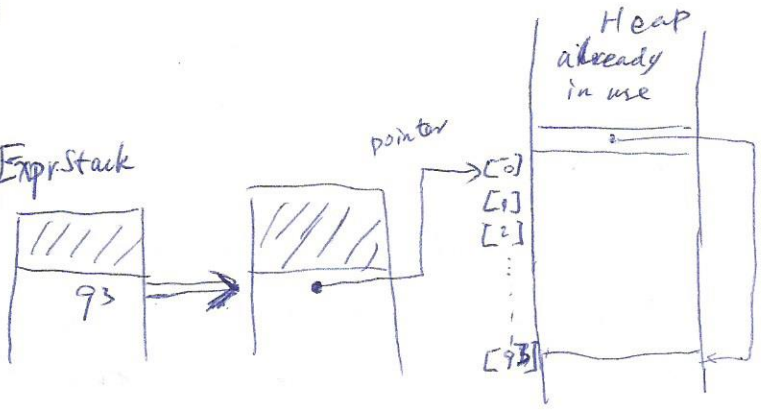
new WRITESTRINGinstr(4, 12) generates 4 12.

@ in ~~<expr2>.whole thing~~

See Parse AND. Translator.java.Txt. <expr2> changes.

CS 316 5/5/2014 ②

new int[93]



Code generated:

<expr3>.code = PUSHNUM 93
HEAPALLOC

while (x < 4) {

/Instruction: getNextCodeAddress()
returns the code memory address of the next instruction to be ~~opp~~ generated

The metod whileStmt() should:

1. Save Instruction.getNextCodeAddress() in an int variable a before calling expr7().
2. Generate the Jumponfalse using something like:

JUMPONFALSEinstr jFinstr = new JUMPONFALSEinstr(Instruction.OPERAND_NOT_YET_KNOWN)

3. After calling statement(), generate

JUMP a

using the variable a of 1.

4. Fixup the JUMPONFALSE using

jFinstr.fixUpOperand(Instruction.getNextCodeAddress())

/AssignmentOr/Invoc.:

x=19; PUSHSTA ADDR 3
PUSHNUM 19
SAVE TO ADDR