MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 76

A Base Language Evaluator for λ-Calculus Expressions

by

D. Austin Henderson, Jr. and S. Nimal Amerasinghe

Work reported herein was supported in part by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number N0014-70-A-0362-0006 and National Science Foundation Contract Number GJ004327. Reproduction in whole or in part is permitted for any purpose of the United States Government.

April 1973

A Base Language Evaluator for λ-Calculus Expressions

by

D. Austin Henderson, Jr. and S. Nimal Amerasinghe

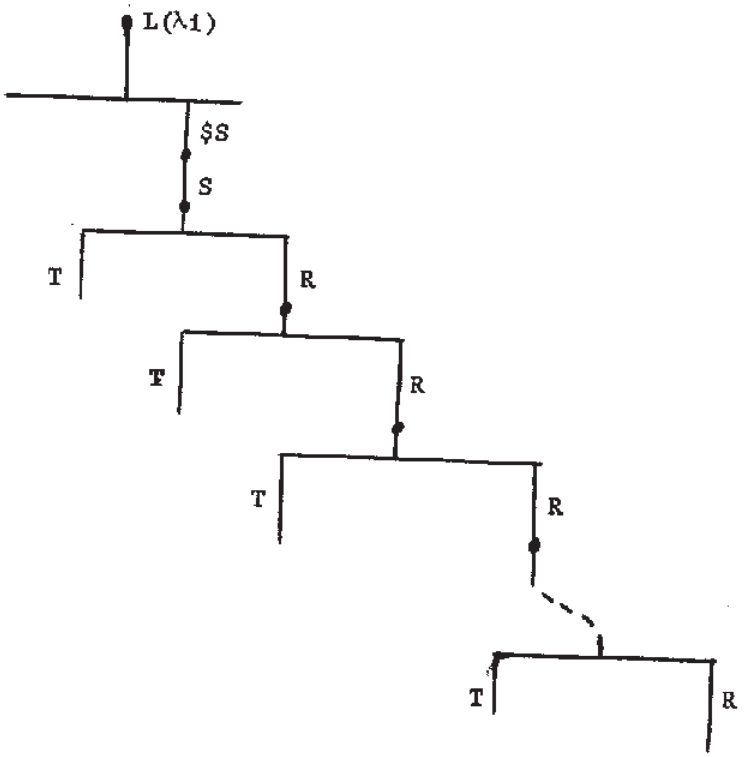
As a sequel to CSG Memo 74 [1], we offer a scheme for translating expressions in the λ-Calculus [2] into procedures structures of the Base Language [3, 4]. The Base Language (BL) interpreter, acting on these control structures, mimics the SECD evaluation [5] of the corresponding λ-Calculus (λC) expression.

See [1] for brief introductory discussions of λC expressions and their order of evaluation.

The Stack

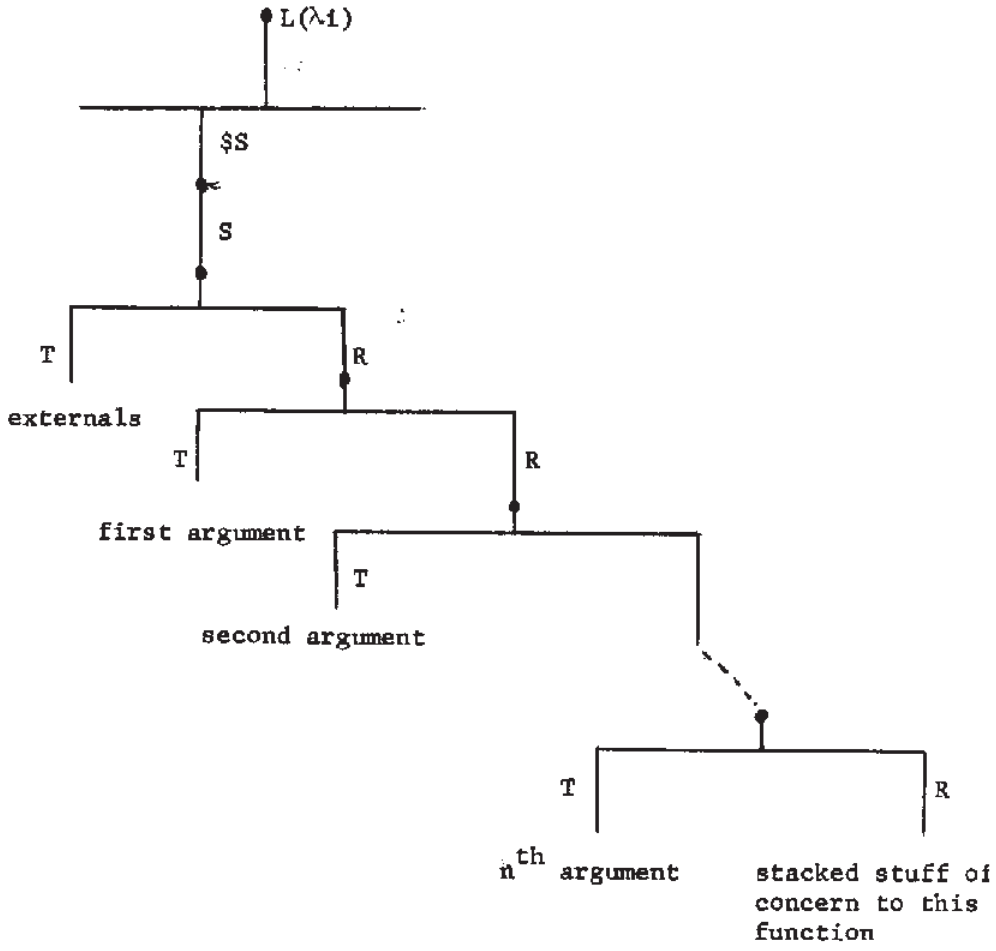
The SECD evaluator is organized around a stack of partially completed results. All computations are carried out from or to this stack. The BL has no such stack, a priori; we therefore establish conventions which maintain a structure fulfilling, by interpretation, the role of the SECD stack.

In each local structure, the stack is selected by $S. It has the following structure:

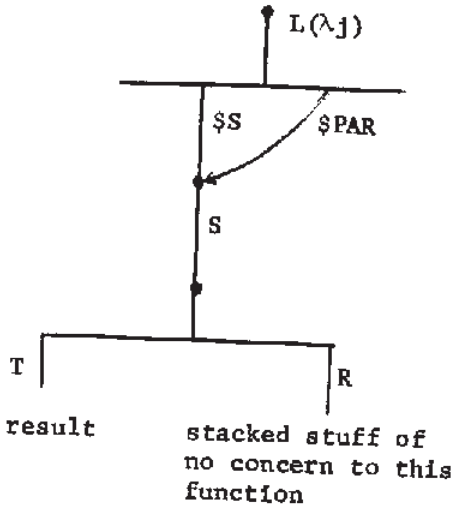


Thus the top-of-stack is selected by $S · S · T; and the n^{th} element on the stack (for top: n = 0) by $S · S · R^n · T.

In the SECD evaluator, each function takes its arguments from the stack, and returns its result by placing it back on the top-of-stack. The BL interpreter passes an argument structure which contains parameters and bindings for external (free) variables of the function being called. We combine these two sets of conventions by placing the external binds on the top-of-stack and passing the resulting stack as the argument structure. Thus just before apply is evaluated the local structure in part looks like:

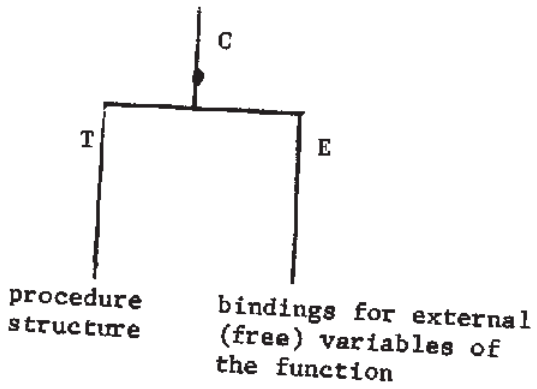


and just before return is executed (in the called function) the local structure in part looks like:



Closures

Functions in the BL are represented by C-structures:



We will use the same structure for representing closures of λ-expressions.

At translation time, the external variables of each λ-expression is known, and so BL instruction can be composed to appropriately determine the bindings of these externals. Specifically when a closure is being formed, the externals are gathered into the E-component of the C-structure; when the closure is called, the E-component is placed on top-of-stack; at the beginning of the code of a function, the E-component (top-of-stack) is torn apart and made part of the local structure of the function, thus providing the external bindings necessary for its evaluation.

Overview of Translation

It is convenient to describe the rules for translating a λC-expression into a BL procedure structure as a three-step process.

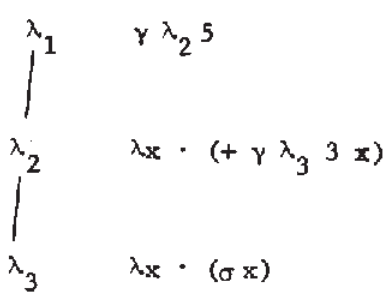
1. Produce a hierarchical structure representing the procedures to be produced and their nesting.
2. Translate the body of each procedure into a sequence of macro calls
3. Expand the macro calls into BL instructions.

Hierarchical Structure (Step 1)

This step is identical to part of the process used in the translation process from λC-expressions to Contour Model algorithms. See Steps 1 and 2 of [1]. It results in a tree of λ-expressions reflecting the nesting of the given λC-expression. The body for each expression is presented near the appropriate node in the tree, and is expressed in Prefix-Polish form with all the apply operators expressed.

Example:

$$\underset{1\ 2}{[}\ [\lambda x \cdot \underset{3}{(+}\ \underset{3}{(\lambda x \cdot \sigma x)3}\ \underset{2\ 1}{x)}]\ 5]$$



Body Translation (Step 2)

The body of each node in the hierarchical structure is translated from Prefix-Polish form to a sequence of calls on a collection of macros. These macros embody the conventions used to mimic to SECD evaluation.

There are two cases: the top level expression, and all other expressions. They differ in that the top-level expression does not need to concern itself with parameters or externals, and needs to "output" a final result rather than "return" it to a caller.

Case 1 (top level): The translation is:

```
crstack                                  ;  create stack initially
⟨translate the body⟩
output                                   ;  "output" the final result
```

Case 2 (λ-expressions): The translation is:

```
setstack                                 ;  access and unstack externals
ext {⟨list of free variables⟩}           ;  select externals
arg                                      ;  unstack and select arguments
⟨translate the body⟩
result is                                ;  return value to caller
```

Common body translation: Translation is done right-to-left on the Prefix-Polish string representing the body. The rules are as follows:

Type of Thing	Translation	Purpose of Macro
constants:	value ⟨constant⟩	; create constant
	push $AC	; stack new value
variable:	push ⟨variable⟩	; stack value of variable
λ-expression:	close ⟨λ-expr⟩, {⟨list of free variables⟩}	; compute a closure
	push $AC	; stack closure
application:	call	; apply a closure
primitive operator:	⟨primitive operator macro⟩	; execute primitive operator

Example: We use the example we started in Step 1. It uses two primitives operators, addition and the successor function, + and σ. The corresponding primitive macros are plus and succ. There are no free variables in this example.

```
λ1:  crstack
     value 5
     push $AC
     close λ2, { }
     push $AC
     call
     output

λ2:  set stack
     ext { }
     arg x
     push x
     value 3
     push $AC
     close λ3, { }
     push $AC
     call
     plus
     resultsis

λ3:  setstack
     ext { }
     arg x
     push x
     succ
     resultis
```

Macro Expansion (Step 3)

The result of Step 2 is a series of macro calls. This step involves expanding these macros into BL instructions. $AC is used to hold temporary results, such as those produced by value and close.

Note that some macros are used in the definition of others. The macro definitions are:

crstack		create	$S · S
		create	$AC
setstack		share	$PAR, $S
		create	$AC
ext $\{e_1, e_2, \ldots, e_n\}$		pop	$E
	for each e_i:	link	$E, e_i, e_i
		delete	$E
arg n		pop	n
resultis		return	
output		pop	$AC
		print	$AC
value n		const	n, $AC
close λ_i, $\{e_i, e_2, \ldots, e_n\}$		delete	$AC
		create	$AC
		move	λ_i, $AC · C · T
		create	$AC · C · E
	for each e_i:	link	$AC · C · E, e_i, e_i
call		select	$S · S, T, $F
		delete	$S · S, T
		link	$S · S, T, $F · C · E
		apply	$F, $S
		delete	$F

```
push n                          select   $S, S, $T
                                delete   $S, S
                                assign   n, $S · S · T
                                link     $S · S, R, $T
                                delete   $T

pop n                           select   $S · S, T, n
                                select   $S · S, R, $T
                                delete   $S, S
                                link     $S, S, $T
                                delete   $T

plus            either:         pop      $A1
                                pop      $A2
                                add      $A1, $A2, $AC
                                push     $AC
                or:             pop      $AC
                                add      $S · S · T, $AC, $S · S · T

succ            either:         pop      $A1
                                add      $A1, 1, $A1
                                push     $A1
                or:             add      $S · S · T, 1, $S · S · T
```

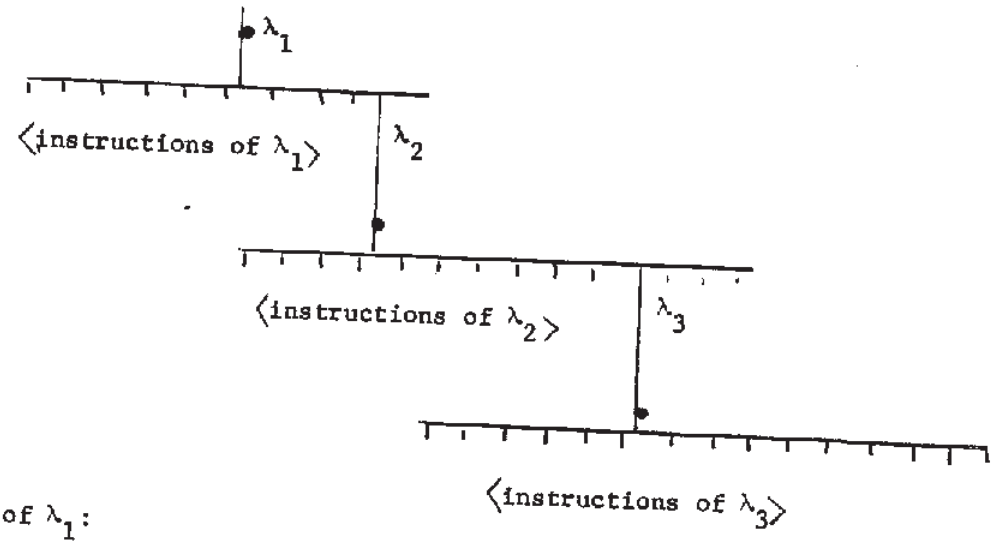
A study of the actions of a program will indicate that these macros do in fact enforce the stack and closure conventions described above.

The share primitive used in setstack is simply a way of renaming the $PAR component of the local structure. Thus all the macros can be written in terms of $S. Of course, $PAR could be substituted every where for $S to achieve the same result. Notice that share is necessary because link provides sharing only at one level removed from the local structure. [Question: Should a generalized share primitive be used to accomplish both these kinds of sharing?].

Note that some care is taken to assure that the temporaries $AC and $T are defined properly for the primitives of the BL as given in [3] and [4].

The two expansions of plus and succ are provided so as 1) to make clear the actions of these macros and 2) be a little less inefficient in accomplishing those actions. The final configurations of the stack are identical.

Example: To complete the example used above the macros are expanded. Integer selectors are added to the instructions generated, and the procedure structures nested as indicated by Step 1, to yield a single translated procedure structure.



Instructions of λ_1:

```
0    create   $S.S
1    create   $AC
3    const    5, $AC
3    select   $S, S, $T
4    delete   $S, S
5    assign   $AC, $S.S.T
6    link     $S.S, R, $T
7    delete   $T
8    delete   $AC
9    create   $AC
10   move     λ2, $AC.C.T
11   create   $AC.C.E
12   select   $S, S, $T
13   delete   $S, S
14   assign   $AC, $S.S.T
15   link     $S.S, R, $T
```

```
; crstack
; value 5
; push $AC
; close λ2, [ ]
; push $AC
```

```
16      delete   $T
17      select   $S·S, T, $F                    ;  call
18      delete   $S·S, T
19      link     $S·S, T, $F·C·E
20      apply    $F, $S
21      delete   $F
22      select   $S·S, T, $AC                   ;  output  ;  pop $AC
23      select   $S·S, R, $T
24      delete   $S, S
25      link     $S, S, $T
26      delete   $T
27      print    $AC                            ;  print $AC
```

Instructions of λ_2:

```
0       share    $PAR, $S                       ;  setstack
1       create   $AC
2       select   $S·S, T, $E                    ;  ext { }  ;  pop $E
3       select   $S·S, R, $T
4       delete   $S, S
5       link     $S, S, $T
6       delete   $T
7       delete   $E                                        ;  delete $E
8       select   $S·S, T, x                     ;  arg x   ;  pop x
9       select   $S·S, R, $T
10      delete   $S, S
11      link     $S, S, $T
12      delete   $T
13      select   $S, S, $T                      ;  push x
14      delete   $S, S
15      assign   x, $S·S·T
16      link     $S·S, R, $T
17      delete   $T
```

```
18      const     3, $AC                          ;   value 3
19      select    $S, S, $T                       ;   push $AC
20      delete    $S, S
21      assign    $AC, $S·S·T
22      link      $S·S, R, $T
23      delete    $T
24      delete    $AC
25      create    $AC                             ;   else λ3, [ ]
26      move      λ3, $AC·C·T
27      create    $AC·C·E
28      select    $S, S, $T                       ;   push $AC
29      delete    $S, S
30      assign    $AC, $S·S·T
31      link      $S·S, R, $T
32      delete    $T
33      select    $S·S, T, $F                     ;   call
34      delete    $S·S, T
35      link      $S·S, T, $F·C·E
36      apply     $F, $S
37      delete    $F
38      select    $S·S, T, $AC                    ;   plus        ;  pop $AC
39      select    $S·S, R, $T
40      delete    $S, S
41      link      $S, S, $T
42      delete    $T
43      add       $S·S·T, $AC, $S·S·T
44      return                                    ;   add
                                                  ;   resultis
```

Instructions of λ_3:

```
 0      share    $PAR, $S                    ;  setstack
 1      create   $AC
 2      select   $S·S, T, $E                 ;  ext { }      ;  pop   $
 3      select   $S·S, R, $T
 4      delete   $S, S
 5      link     $S, S, $T
 6      delete   $T
 7      delete   $E                                          ;  delete $
 8      select   $S·S, T, x                  ;  arg x        ;  pop x
 9      select   $S·S, R, $T
10      delete $S, S
11      link     $S, S, $T
12      delete   $T
13      select   $S, S, $T                   ;  push x
14      delete   $S, S
15      assign   x, $S·S·T
16      link     $S·S, R, $T
17      delete   $T
18      add      $S·S·T, 1, $S·S·T           ;  succ
19      return                               ;  resultis
```

Efficiency

A study of the above BL code reveals that a certain amount of optimization is possible. For example, instructions 7 through 14 may be deleted from λ_2 and 7 through 14 from λ_3. Both sets implement a "pop x" followed by a "push x", and as x is only used once in each piece of code there is no need to have x ever appear in the local structure. However, the translation rule is quite general, and we are not writing an optimizing translator (compiler). Such non-optimal code may be quite common; this does not concern us here.

More Complex Examples

The reader may wish to check that this translation scheme is successful for some more complex examples. The following examples are used in [1], and afford interesting three-way comparisons between the λC, BL, and CM.

$$\{[\lambda f \cdot (\lambda f \cdot f3)(\lambda x \cdot (fx) + 2)][\lambda x \cdot x + 1]\}$$

$$\{[\lambda g \cdot (\lambda a \cdot (ga)5)\ 4][\lambda y \cdot [\lambda a \cdot (\lambda g \cdot g)(\lambda x \cdot x + a + y)]\ 3]\}$$

Recursion

The BL is neither block structured nor "inherently" recursive. Explicit actions necessary to translate recursive programs so that references are correct; namely, that a C-structure can be referred to by the text of the function which is its T-component. Two means are possible for achieving this: some identifier in the E-component can references the C-structure (thus introducing a cycle to the interpreter state), or the C-structure can be passed as an argument to the function. In either case, the C-structure is accessible from the local structure of the activation of the function, which means that it can be invoked recursively.

As pointed out in [1], there is no need to deal specially with recursion when translating λC-expressions for a recursion-making operator, Y, can be expressed as a λC-expression itself. The rules we have will thus permit recursion as they stand. However, they do so in a complex way including creation at every level of recursion of new C-structures which represent "the same function." See [6].

We can bypass this circumlocation by giving an explicit rule for translating $Y(\lambda f \cdot \lambda x \cdot \text{———})$.

Translation of $Y(\lambda v \cdot \lambda x \cdot \text{———})$

Step 0: In the original λC-expression, replace

$$Y(\lambda v \cdot \lambda x \cdot \text{———————})$$

with

$$R_v(\lambda x \cdot \text{———————})$$

Step 1: Same as regular Step 1 with R_v treated as a special control symbol.

Step 2: Same as regular Step 2 except that there is a special case for translating λ-expressions when the λ_i is preceded by R_v.

$R_v\ \lambda_i$ translates to

```
rclose λi, v, [⟨list of free variables⟩]
push   $AC
```

Step 3: Same as regular Step 3 with the following additional macro:

rclose λ_i, v, $\{e_1, \ldots, e_n\}$ expands to

```
close  λi, {e1, ..., en}
link   $AC·C·E, v, $AC
```

Example: Consider the recursive factorial function as given in [1]:

$$[[\lambda f \cdot f3][Y(\lambda f \cdot [\lambda x \cdot (\text{if } x \le 1 \text{ then } 1 \text{ else } f(x-1)*x)])]]$$

becomes

$$\{\ [\lambda f \cdot f3]\ [R_f\ [\lambda x \cdot (\text{if } x \le 1 \text{ then } 1 \text{ else } f(x-1)*x)]]\ \}$$

λ_1 $\quad Y\ \lambda_2\ R_f\ \lambda_3$

λ_2 $\quad \lambda f \cdot Y\ f3$

λ_3 $\quad \lambda x \cdot$ if $x \le 1$ then 1

else * Yf - x 1 x

For translation's sake, we re-write λ_3 as

$$\lambda x \cdot \delta_1 \delta_2 \beta \leq x\ 1$$

where

$$\delta_1 = 1$$

$$\delta_2 = *\gamma f - x\ 1\ x$$

λ_1 translates:

```
crstack
rclose λ3, f, { }
close λ2
call
output
```

λ_2 translates: as before

λ_3 translates:

```
setstack
ext {f}
arg x
value 1
push $AC
push x
le                          ;  less than or equal
test δ1, δ2
resultis
```

δ_1 translates:

```
value 1
push $AC
```

δ_2 translates:

```
push x
value 1
push $AC
push x
minus
push f
call
multiply
```

The expansions are lengthy and straightforward and so are omitted. The expansion of test includes compiling the obvious jumps to include the code for δ_1 and δ_2, as follows:

test δ_i, δ_j	k	pop	$AC		
	k+4	if	$AC	goto	ℓ+1
	k+5	⟨expansion of δ_i⟩			
	ℓ	goto	m		
	ℓ+1	⟨expansion of δ_j⟩			
	m				

Conclusion

This memo gives an algorithm for translating λ-calculus expressions into procedure structures for the Base Language so that the BL interpreter simulates the SECD evaluation of the λC-expressions. The algorithm depends on supplying names for the λ-expressions in the λC-expressions.

Although recursion is provided for by the λ-calculus itself, it is more appealing to provide for it explicitly in the Base Language. Consequently, a special rule for translating "Y(λf . λx . ————)" is given which yield algorithms causing the Base Language interpreter to achieve the recursion effect of the λ-calculu "Y" operator.

References

1. Amerasinghe, S. N. and Henderson, D. A., Jr., A Contour Model Evaluator for λ-Calculus Expressions. Computation Structures Group Memo 74, Project MAC, M.I.T., February 1973.

2. Church, A., The calculi of lambda conversion. Annals of Mathematical Study, No. 6, Princeton, New Jersey, 1941.

3. Dennis, J. B., On the Design and Specification of a Common Base Language. Computation Structures Group Memo 60, Project MAC, M.I.T., July 1971.

4. Amerasinghe, S. N., The Handling of Procedure Variables in a Base Language. S.M. Thesis, Department of Electrical Engineering, M.I.T., September 1972.

5. Landin, P. J., The mechanical evaluation of expressions. Computer Journal, Vol. 6, No. 4, January 1964.

6. Wozencraft, J. M. and Evans, A., Jr., Notes on Programming Linguistics. Notes for Course 6.231, Department of Electrical Engineering, M.I.T., 1970.