This code appears in the class directory. It is provided here with color highlighting to aid the identification of the changes. The changed portions of the code appear in red.
% The following is BNF to describe a simple collection of expressions.
% Note that for simplicity in this BNF, no constants or whitespace are
% provided for, and only the letters 'a' through 'z' can be used in
% identifiers. A production label denotes the kind of expr produced by
% using it, and these labels appear as node names in derivation trees.

% To have lowest precedence, the conditional will appear at the top of
% derivation trees. A new start symbol <cond> is introduced to accomplish
% this. Other operations can then be introduced only AFTER the conditional.

% exp: <cond> --> <expr>  cnd: <cond> --> <expr>?<cond>:<cond>
% trm: <expr> --> <term>  ide: <factor> --> <ident>
% sum: <expr> --> <term><term>  prn: <factor> --> (<cond>)
% dif: <expr> --> <factor><term>  let: <ident> --> <letter>
% fac: <term> --> <factor>  idL: <ident> --> <ident> <letter>
% prd: <term> --> <term>*<factor>  a: <letter> --> a ...
% div: <term> --> <term>/<factor>  z: <letter> --> z

% To evaluate an expression E (entered as a string), an environment must be
% given -- an environment will just be a list of pairs of identifiers and
% their numeric values of the form
% [(id1,n1), (id2,n2), . . . , (idk,nk)] - ids should be atoms not strings.
% evaluate(Expr, Env, Ans) succeeds with Ans as the value of Expr given Env
evaluate(Expr, Env, Ans) :-
   parse_cond(Expr, Dtree),  % add write(Dtree) here to see derivation
   compile(Dtree, Code),    % add write(Code) here to see instructions
   execute(Code,0,Env,Ans).  % initialize accum. to 0

% Analyze strings and construct a derivation tree according to the BNF above.
% Each clause corresponds to one of the productions -- the first argument
% is the string to be parsed, and the second is its tree with the production
% labels giving the root type.

parse_cond(Xs, exp(T)) :- parse_expr(Xs, T).
parse_cond(Xs, cnd(T1,T2,T3)) :- name('?', [N1]), append(Ys, [N1|Zs], Xs),
    name(':', [N2]), append(Vs, [N2|Ws], Zs), parse_expr(Ys, T1),
    parse_cond(Vs, T2), parse_cond(Ws, T3).
parse_expr(Xs, trm(T)) :- parse_term(Xs, T).
parse_expr(Xs, sum(T1,T2)) :- name('+', [N]), append(Ys, [N|Zs], Xs),
    parse_expr(Ys, T1), parse_term(Zs, T2).
parse_expr(Xs, dif(T1,T2)) :- name('-', [N]), append(Ys, [N|Zs], Xs),
    parse_expr(Ys, T1), parse_term(Zs, T2).

parse_term(Xs, fac(T)) :- parse_factor(Xs, T).
parse_term(Xs, prd(T1,T2)) :- name('*', [N]), append(Ys, [N|Zs], Xs),
    parse_term(Ys, T1), parse_factor(Zs, T2).
parse_term(Xs, div(T1,T2)) :- name('/', [N]), append(Ys, [N|Zs], Xs),
    parse_term(Ys, T1), parse_factor(Zs, T2).
parse_factor(Xs, ide(T)) :- parse_ident(Xs, T).
parse_factor(Xs, prn(T)) :- name('(', [N1]), name(')'), N2,
                  append([N1|Ys], N2, Xs), parse_cond(Ys, T).

parse_ident([X], let(T)) :- parse_letter([X], T).
parse_ident(Xs, idL(T1,T2)) :-
            append(Ys, [X], Xs), parse_ident(Ys, T1),
            parse_letter([X], T2).

parse_letter([X],L) :-
            97=<X, X=<122, name(L,[X]).

% The following predicate will accept a derivation tree from parse, and
% produce a list of "assembly language" instructions to evaluate the expr.
% The assembler uses only nine instructions: load accum. from memory, store
% accum. into memory, add memory to accum., subtract memory from accum.
% multiply accum. by memory, divide accum. by memory, a no-op,
% branch to (i.e., goto) the instruction N (the address) forward from the
% current one, and branch on zero accum. to the instruction N forward from
% current one.

% Code for any (sub) expression leaves its value in the accumulator.
% A no-op instruction needs to be added as a last instruction as an exit
% target in case the last sub-expression is a conditional

compile(Tree, Code) :-
    compile(Tree, Code1, 1), append(Code1, [noop], Code).
% The third argument N is the index of the next available temporary memory
% location to be used.
compile(exp(T), Code, N) :- compile(T, Code, N).
% The code for the conditional is
% code(e1?e2:e3) = code(e1)++[brz(N1)]++code(e2)++[br(N2)]++code(e3)
% where N1 branches over code(e2) and N2 branches over code(e3)
compile(cnd(T1,T2,T3), Code, N) :-
    compile(T1, Code1, N), compile(T2, Code2, N), compile(T3, Code3, N),
    length(Code2, L2), B1 is L2+2,
    length(Code3, L3), B2 is L3+1,
    append(Code2, [brz(B1)|Code3], Code23),
    append(Code1, [brz(B2)|Code23], Code).

compile(symb(T), Code, N) :-
    compile(symb(T), Code, N).
% N is temp memory index
compile(sum(T1,T2), Code, N) :-
    compile(T2, Code2, N),
    append(Code2, [sto(Temp) | Code1], Code3),
    append(Code3, [add(Temp)], Code).

compile(dif(T1,T2), Code, N) :-
    compile(T2, Code2, N),
    append(Code2, [sto(Temp) | Code1], Code3),
    append(Code3, [sub(Temp)], Code).

compile(fac(T), Code, N) :- compile(T, Code, N).
compile(prd(T1,T2), Code, N) :-
    compile(T2, Code2, N), N1 is N+1, compile(T1, Code1, N1),
    nextTemp(Temp,N),
    append(Code2, [sto(Temp) | Code1], Code3),
    append(Code3, [mul(Temp)], Code). % 'mul' is multiply inst

compile(div(T1,T2), Code, N) :-
    compile(T2, Code2, N), N1 is N+1, compile(T1, Code1, N1),
    nextTemp(Temp,N),
    append(Code2, [sto(Temp) | Code1], Code3),
    append(Code3, [div(Temp)], Code). % 'div' is division inst

compile(prn(T), Code, N) :- compile(T, Code, N).

compile(ide(T), [ld(Id)], N) :- compile(T, Id1, N), name(Id, Id1).
% load inst

compile(idL(T,L), Id, N) :- compile(T, Id1, N), name(L,X), append(Id1, X, Id).

compile(let(L), X, _) :-
    name(L,X).
% nextTemp(-Var,+N) builds a temporary identifier, Var=tempN

nextTemp(Temp,N) :- name(temp,T), name(N,X), append(T,X,TX), name(Temp,TX).
% The predicate 'execute' accepts code from compile, an initial value
% for the arithmetic register, and list of values for the identifiers
% appearing in the expression (an environment), and executes the instructions
% for the virtual-machine to produce the value of the original expression.

execute(Code, Accum, Env, Result) :-
    append(Env, _, Env_), execute2(Code, Accum, Env_, Result).

execute2([], R, _, R). % end of execution, Result=Register

execute2([noop | Code], Reg, Env, Result) :-
    execute2(Code, Reg, Env, Result).

execute2([ld(A) | Code], _, Env, Res) :-
    member((A,X), Env)  % lookup value of A in Env
    -> execute2(Code,X,Env,Res)
    ; undefined_var(A).

execute2([sto(A) | Code], Reg, Env, Res) :-
    append(Env1, [(A,_) | Env2], Env) ->  % decompose Env into pieces
    append(Env1, [(A,Reg) | Env2], Env3), % place new value of A in Env3
    execute2(Code, Reg, Env3, Res).
execute2([add(A) | Code], Reg, Env, Res) :-
    member((A,X), Env) % lookup value of A in Env
    -> Y is Reg+X, execute2(Code, Y, Env, Res)
    ; undefined_var(A).
execute2([sub(A) | Code], Reg, Env, Res) :-
    member((A,X), Env) % lookup value of A in Env
    -> Y is Reg-X, execute2(Code, Y, Env, Res)
    ; undefined_var(A).
execute2([mul(A) | Code], Reg, Env, Res) :-
    member((A,X), Env) % lookup value of A in Env
    -> Y is Reg*X, execute2(Code, Y, Env, Res)
    ; undefined_var(A).
execute2([div(A) | Code], Reg, Env, Res) :-
    member((A,X), Env) % lookup value of A in Env
    -> Y is Reg/X, execute2(Code, Y, Env, Res)
    ; undefined_var(A).
execute2([br(A) | Code], Reg, Env, Res) :-
    A1 is A-1,
    drop(A1,Code,Code1),
    execute2(Code1, Reg, Env, Res).
execute2([brz(A) | Code], Reg, Env, Res) :-
    A1 is A-1,
    drop(A1,Code,Code1),
    Reg =:= 0
    -> execute2(Code1, Reg, Env, Res)
    ; execute2(Code, Reg, Env, Res).

undefined_var(A) :- write(undefined_var(A)), nl, fail.

% drop(?N,?Xs,?Ys) succeeds when Ys is the list obtained by removing the first
% N items of list Xs.
drop(N,Xs,Ys) :- append(Zs,Ys,Xs), length(Zs,N).