Semantics of the Source Language

Source Language:

```
A ::= n
   |  x
   |  add(A,A)
   |  sub(A,A)
   |  mult(A,A)

B ::= true
   |  false
   |  eq(A,A)
   |  le(A,A)
   |  not(B)
   |  and(B,B)
   |  or(B,B)

C ::= skip
   |  assign(x,A)
   |  seq(C,C)
   |  if(B,C,C)
   |  whiledo(B,C)
```

Semantics of the Source Language

Semantics are the same compared to our initial simple imperative language; arithmetic expressions:

```
(C,_) -->> C :-                % constants
    int(C),!.

(X,State) -->> Val :-          % variables
    atom(X),
    lookup(X,State,Val),!.

(add(A,B),State) -->> Val :-   % addition
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis ValA + ValB,!.

(sub(A,B),State) -->> Val :-   % subtraction
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis ValA - ValB,!.

(mult(A,B),State) -->> Val :-  % multiplication
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis ValA * ValB,!.
```

Semantics of the Source Language

Boolean expressions:

```
(true,_) -->> true :- !.                % constants

(false,_) -->> false :- !.              % constants

(eq(A,B),State) -->> Val :-             % equality
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis (ValA =:= ValB),!.

(le(A,B),State) -->> Val :-             % le
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis (ValA =< ValB),!.

(not(A),State) -->> Val :-              % not
    (A,State) -->> ValA,
    Val xis (not ValA),!.

(and(A,B),State) -->> Val :-            % and
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis (ValA and ValB),!.

(or(A,B),State) -->> Val :-             % or
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis (ValA or ValB),!.
```

Semantics of the Source Language

Statements:

```
(skip,State) -->> State :- !.          % skip

(assign(X,A),State) -->> OState :-     % assignment
    (A,State) -->> ValA,
    put(X,ValA,State,OState),!.

(seq(C0,C1),State) -->> OState :-      % composition, seq
    (C0,State) -->> S0,
    (C1,S0) -->> OState,!.

(if(B,C0,_),State) -->> OState :-     % if
    (B,State) -->> true,
    (C0,State) -->> OState,!.

(if(B,_,C1),State) -->> OState :-     % if
    (B,State) -->> false,
    (C1,State) -->> OState,!.

(whiledo(B,_),State) -->> OState :-    % while
    (B,State) -->> false,
    State=OState,!.

(whiledo(B,C),State) -->> OState :-    % while
    (B,State) -->> true,
    (C,State) -->> SC,
    (whiledo(B,C),SC) -->> OState,!.
```

Semantics of the Target Language

Target Language:

```
prog ::= [ cmseq ]   |  [ ]

cmseq ::=  cm   |  cm , cmseq

cm ::= push(V)
     |  add
     |  sub
     |  mult
     |  and
     |  or
     |  neg
     |  eq
     |  le
     |  pop(x)
     |  label(L)
     |  jmp(L)
     |  jmpt(L)
     |  jmpf(L)
     |  stop

V ::=  x  |  n  | true | false

L ::=  <alpha string>
```

A state in our machine is a term of arity two where the first component is an integer stack used for expression evaluation and the second component is a binding environment for variables:

'(Stack,Environment)'

Semantics of the Target Language

Flow of control instructions: perhaps the most surprising part of the semantics for our target language is the notion of a *continuation*. A continuation is a way to model the address space of the target machine so that we can perform *jumps* to labels.

```
% the predicate '(+Syntax,+Continuation,+State) -->> -State' computes
% the semantic value for each syntactic structure

([],_,State) -->> State :- !.          % an empty instruction sequence is a noop

([stop|_],_,State) -->> State :- !.    % the 'stop' instruction ignores the rest of the program

([jmp(L)|_],Cont,State) -->> OState :-
    afindlabel(L,Cont,JT),
    (JT,Cont,State) -->> OState,!.

([jmpt(_)|P],Cont,([false|Stk],Env)) -->> OState :-
        (P,Cont,(Stk,Env)) -->> OState,!.

([jmpt(L)|_],Cont,([true|Stk],Env)) -->> OState :-
         afindlabel(L,Cont,JT),(JT,Cont,(Stk,Env)) -->> OState,!.

([jmpf(L)|_],Cont,([false|Stk],Env)) -->> OState :-
         afindlabel(L,Cont,JT),(JT,Cont,(Stk,Env)) -->> OState,!.

([jmpf(_)|P],Cont,([true|Stk],Env)) -->> OState :-
        (P,Cont,(Stk,Env)) -->> OState,!.
```

Semantics of the Target Language

A continuation is a copy of the original program and we use it to look up jump targets:

```
% the predicate 'afindlabel(+Label,+Continuation,-JumpTarget)'
% looks up a label definition in the continutation and returns its associate code.
:- dynamic afindlabel/3.

afindlabel(L,[label(L)|P],[label(L)|P]).

afindlabel(L,[_|P],JT) :-
    afindlabel(L,P,JT).

afindlabel(_,[],_) :-
   writeln('ERROR: label not found.'),!,fail.
```

Semantics of the Target Language

Computational instructions:

```
%%% computational instructions
([Instr|P],Cont,State) -->> OState :-  % interpret an instruction sequence.
    (Instr,Cont,State) -->> IState,
    (P,Cont,IState) -->> OState,!.

(push(C),_,(Stk,Env)) -->> ([C|Stk],Env) :-        % constants
    int(C),!.

(push(X),_,(Stk,Env)) -->> ([ValX|Stk],Env) :-     % variables
    atom(X),
    alookup(X,Env,ValX),!.

(pop(X),_,([ValA|Stk],Env)) -->> (Stk,OEnv) :-  % store
    aput(X,ValA,Env,OEnv),!.

(add,_,([ValB,ValA|Stk],Env)) -->> ([Val|Stk],Env) :-  % addition
    Val xis ValA + ValB,!.

...

(and,_,([ValB,ValA|Stk],Env)) -->> ([Val|Stk],Env) :-  % and
    Val xis (ValA and ValB),!.

...

(neg,_,([ValA|Stk],Env)) -->> ([Val|Stk],Env) :-  % not
    Val xis (not ValA),!.

(label(_),_,State) -->> State :- !.
```

Semantics of the Target Language

Interpreting the target language in its model,

```
?- ['target.pl'].
%  xis.pl compiled 0.00 sec, 6,824 bytes
% target.pl compiled 0.00 sec, 14,600 bytes
true.

?- assert(program([push(1),push(2),add])).
true.

?- program(P), (P,P,([],e)) -->> S.
P = [push(1), push(2), add],
S = ([3], e).

?-
```

The `P` in red is the continuation.