Compiler Correctness

The idea of a correct compiler is that the intended behavior of the source program is preserved by the translated program.

Compiler Correctness

Recall that `translate(c,c')` is our predicate that takes a source program `c` and produces the target code `c'`.

```
:- >>> 'executing: assign(x,2) seq assign(y,mult(2,x))'.

program1(assign(x,2) seq assign(y,mult(2,x))).

:- >>> 'Source Semantics'.
:- program1(P),(P,e) -->> Env, Env = env([bind(4,y)|_],e).

:- >>> 'Target Semantics'.
:- program1(P),translate(P,C),(C,C,([],e)) -->> ([],Env), Env = env([bind(4,y)|_],e).
```

Compiler Correctness

```
:- >>> 'executing: if(le(2,3),assign(i,3),assign(i,4))'.

program2(if(le(2,3),assign(i,3),assign(i,4))).

:- >>> 'Source Semantics'.
:- program2(P),(P,e)  -->> Env,  Env = env([bind(3,i)|_],e).

:- >>> 'Target Semantics'.
:- program2(P),translate(P,C),(C,C,([],e)) -->> ([],Env), Env = env([bind(3,i)|_],e).
```

Compiler Correctness

```
:- >>> 'executing: '.
:- >>> 'assign(n,3) seq'.
:- >>> 'assign(i,1) seq'.
:- >>> 'assign(z,1) seq'.
:- >>> 'whiledo(not(eq(i,n)),'.
:- >>> '   assign(i,add(i,1)) seq'.
:- >>> '   assign(z,mult(z,i)))'.

program3(assign(n,3) seq
         assign(i,1) seq
         assign(z,1) seq
         whiledo(not(eq(i,n)),
           assign(i,add(i,1)) seq
           assign(z,mult(z,i)))).

:- >>> 'Source Semantics'.
:- program3(P),(P,e) -->> Env, Env = env([bind(6,z)|_],e).

:- >>> 'Target Semantics'.
:- program3(P),translate(P,C),(C,C,([],e)) -->> ([],Env), Env = env([bind(6,z)|_],e).
```

Compiler Correctness

The compiler correctness problem is a special case of our *semantic equivalence* problem.

Consider the case of translating statements, then for any $c \in \textbf{Com}$ and its corresponding translated code c' with $\mathrm{translate}(c, c')$,

$$c \sim c' \text{ iff } \forall e, \exists E[(c, e) \longrightarrow\!\!\!\gg E \wedge (c', ([\,], e)) \longrightarrow\!\!\!\gg ([\,], E)]$$

Compiler Correctness

Similarly, the case of translating arithmetic expressions, then for any $a \in$ **Aexp** and its corresponding translated code a' with $\mathrm{translate}(a, a')$,

$$a \sim a' \text{ iff } \forall e, \exists V[(a, e) \longrightarrow\!\!\!\gg V \wedge (a', ([\,], e)) \longrightarrow\!\!\!\gg ([V], e)]$$

Compiler Correctness

And finally, the case of translating boolean expressions, then for any $b \in \textbf{Bexp}$ and its corresponding translated code b' with $\mathrm{translate}(b, b')$,

$$b \sim b' \text{ iff } \forall e, \exists B, B' [(b, e) \longrightarrow\!\!\!\gg B \wedge (b', ([\,], e)) \longrightarrow\!\!\!\gg ([B'], e) \wedge B' = B]$$

Compiler Correctness

The correctness proofs themselves are proofs by induction. There is no problem here because translation is a syntactic property – we are NOT trying to prove an algorithm written in the source language correct – we are trying to prove that whatever algorithm was written in the source language (correct or not) is preserved by the translation.

Compiler Correctness: Booleans

```
% it is sufficient to show that for every b in Bexp we have
%
%  (forall b,e)[(b,e) -->> V1 ^
%                translate(b,C)  ^
%                (C,C,([],e)) --> ([V2],e) ^
%                V1 = V2]
%
% proof by induction on Bexp.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- >>> 'case true'.
:- (true,e) -->> V1,
   translate(true,C),
   (C,C,([],e)) -->> ([V2],e),
   V1 = V2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- >>> 'case false'.
:- (false,e) -->> V1,
   translate(false,C),
   (C,C,([],e)) -->>([V2],e),
   V1 = V2.
```

Compiler Correctness: Booleans

```
% the following assumptions hold for the relational operators
:- asserta((a,e) -->> va).  % arithmetic exp a -> va
:- asserta((b,e) -->> vb).  % arithmetic exp b -> vb
:- asserta(int(va)).
:- asserta(int(vb)).
:- asserta(translate(a,ca)).
:- asserta(translate(b,cb)).
:- asserta((ca,_,(S,E)) -->> ([va|S],E)).
:- asserta((cb,_,(S,E)) -->> ([vb|S],E)).

:- >>> 'case eq(a,b)'.
:- (eq(a,b),e) -->> V1,
   translate(eq(a,b),C),
   (C,C,([],e))  -->> ([V2],e),
   V1 = V2.

:- >>> 'case le(a,b)'.
:- (le(a,b),e) -->> V1,
   translate(le(a,b),C),
   (C,C,([],e)) -->> ([V2],e),
   V1 = V2.

:- retract((a,e) -->> va).
:- retract((b,e) -->> vb).
:- retract(int(va)).
:- retract(int(vb)).
:- retract(translate(a,ca)).
:- retract(translate(b,cb)).
:- retract((ca,_,(S,E)) -->> ([va|S],E)).
:- retract((cb,_,(S,E)) -->> ([vb|S],E)).
```

Compiler Correctness: Booleans

```
:- asserta((a,e) -->> va).
:- asserta((b,e) -->> vb).
:- asserta(bool(va)).
:- asserta(bool(vb)).
:- asserta(translate(a,ca)).
:- asserta(translate(b,cb)).
:- asserta((ca,_,(S,E)) -->> ([va|S],E)).
:- asserta((cb,_,(S,E)) -->> ([vb|S],E)).

:- >>> 'case not(a)'.
:- (not(a),e) -->> V1,
    translate(not(a),C),
    (C,C,([],e)) -->> ([V2],e),
    V1 = V2.

:- >>> 'case and(a,b)'.
:- (and(a,b),e) -->> V1,
    translate(and(a,b),C),
    (C,C,([],e)) -->> ([V2],e),
    V1 = V2.

:- >>> 'case or(a,b)'.
. . .

:- retract((a,e) -->> va).
:- retract((b,e) -->> vb).
:- retract(bool(va)).
:- retract(bool(vb)).
:- retract(translate(a,ca)).
:- retract(translate(b,cb)).
:- retract((ca,_,(S,E)) -->> ([va|S],E)).
:- retract((cb,_,(S,E)) -->> ([vb|S],E)).
```

Compiler Correctness: Commands

```
% it is sufficient to show that for every c in Com we have
%
%  (forall cc,e)[(cc,e) -->> E1) ^
%              translate(cc,C) ^
%              (C,C,([],e)) -->> ([],E1))]
%
% proof by induction on Com.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- >>> 'case skip'.
:- (skip,e) -->> E1,
    translate(skip,C),
    (C,C,([],e)) -->> ([],E1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- >>> 'case assign'.
:- asserta((a,e) -->> va).
:- asserta(translate(a,ca)).
:- asserta((ca,_,(L,S)) -->> ([va|L],S)).

:- (assign(x,a),e) -->> E1,
   translate(assign(x,a),C),
   (C,C,([],e)) -->> ([],E1).

:- retract((a,e) -->> va).
:- retract(translate(a,ca)).
:- retract((ca,_,(L,S)) -->> ([va|L],S)).
```

Compiler Correctness: Commands

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- >>> 'case seq'.
:- asserta((c1,_) -->> e1).
:- asserta((c2,e1) -->> e2).
:- asserta(translate(c1,cc1)).
:- asserta(translate(c2,cc2)).
:- asserta((cc1,_,([],_)) -->> ([],e1)).
:- asserta((cc2,_,([],e1)) -->> ([],e2)).

:- (seq(c1,c2),e) -->> E1,
    translate(seq(c1,c2),C),
    (C,C,([],e)) -->> ([],E1).

:- retract((c1,_) -->> e1).
:- retract((c2,e1) -->> e2).
:- retract(translate(c1,cc1)).
:- retract(translate(c2,cc2)).
:- retract((cc1,_,([],_)) -->> ([],e1)).
:- retract((cc2,_,([],e1)) -->> ([],e2)).
```

Compiler Correctness: Commands

```
:- >>> 'case if'.
:- asserta((c1,e) -->> e1).
:- asserta((c2,e) -->> e2).
:- asserta(translate(b,cb)).
:- asserta(translate(c1,cc1)).
:- asserta(translate(c2,cc2)).
:- asserta((cc1,_,([],e)) -->> ([],e1)).
:- asserta((cc2,_,([],e)) -->> ([],e2)).

:- >>> '    case analysis, b=true'.
:- asserta((b,e) -->> true).
:- asserta((cb,_,([],e)) -->> ([true],e)).

:- (if(b,c1,c2),e) -->> E1,
    translate(if(b,c1,c2),C),
    (C,C,([],e)) -->> ([],E1).

:- retract((b,e) -->> true).
:- retract((cb,_,([],e)) -->> ([true],e)).

:- >>> '    case analysis, b=false'.
:- asserta((b,e) -->> false).
:- asserta((cb,_,([],e)) -->> ([false],e)).

:- (if(b,c1,c2),e) -->> E1,
    translate(if(b,c1,c2),C),
    (C,C,([],e)) -->> ([],E1).

:- retract((b,e) -->> false).
:- retract((cb,_,([],e)) -->> ([false],e)).
...
```

Compiler Correctness: Commands

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- >>> 'case while'.
:- asserta((c,e) -->> vc).
:- asserta(translate(b,cb)).
:- asserta(translate(c,cc)).
:- asserta((cc,_,([],e)) -->> ([],vc)).

:- >>> '    case analysis, b=false'.
:- asserta((b,e) -->> false).
:- asserta((cb,_,([],e)) -->> ([false],e)).

:- (whiledo(b,c),e) -->> e,
    translate(whiledo(b,c),C),
    (C,C,([],e)) -->> ([],e).

:- retract((b,e) -->> false).
:- retract((cb,_,([],e)) -->> ([false],e)).
```

Compiler Correctness: Commands

Here we prove that one iteration of the loop executes correctly, subsequent iterations of the loop are assumed to terminate in some state vt.

```
:- >>> '    case analysis, b=true'.
:- asserta((b,e) -->> true).
:- asserta((cb,_,([],e)) -->> ([true],e)).

% lemma: prove that the translation of whiledo(b,c) is the piece of stack machine code shown
:- translate(
    whiledo(b,c),
    [label(whilelabel1),cb,jmpf(whilelabel2),cc,jmp(whilelabel1),label(whilelabel2)]).

% it is sufficient to prove the correctness for the loop for only one iteration
% assume that the remaining itereations after the first iteration give us some
% terminal state vt - we assume this both in the source and target language
:- asserta((whiledo(b,c),vc) -->> vt).
:- asserta(([label(whilelabel1),cb,jmpf(whilelabel2),cc,jmp(whilelabel1),label(whilelabel2)],
                _,([],vc)) -->> ([],vt)).

:- (whiledo(b,c),e) -->> vt,
    translate(whiledo(b,c),C),
    (C,C,([],e)) -->> ([],vt).

:- retract((b,e) -->> true).
:- retract((cb,_,([],e)) -->> ([true],e)).
:- retract((whiledo(b,c),vc) -->> vt).
:- retract(([label(whilelabel1),cb,jmpf(whilelabel2),cc,jmp(whilelabel1),label(whilelabel2)],
                _,([],vc)) -->> ([],vt)).
```

Assignment

Assignment #6 – see website