Translational Semantics

What if we define the semantics of some language in terms of another language by translating our source language syntax into the syntax of some target language?

In this case we obtain the following diagram,

$$
\begin{tikzcd}
\mathrm{Model}_{\mathrm{Source}} \arrow[rr, phantom, "\approx"] & & \mathrm{Model}_{\mathrm{Target}} \\
\mathrm{Syntax}_{\mathrm{Source}} \arrow[u, "\text{interpretation}"] \arrow[rr, "\text{translation}"'] & & \mathrm{Syntax}_{\mathrm{Target}} \arrow[u, "\text{interpretation}"']
\end{tikzcd}
$$

Translational Semantics

Two Observations:

1. This means we have now two ways to interpret a sentence in the source language:
   1. we can interpret the sentence in the source language model
   2. we can first translate the sentence and then interpret it in the model of the target language
2. We say that the translation is *correct* if we can establish a correspondence between the source and target models for each source language syntax structure.

Translational Semantics

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)
```

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>
```

Translational Semantics

We can define a semantics for our source language by defining equivalent sentences in the target language for each syntactic construct in the source language. Consider the arithmetic expressions:

```
translate(N,[push(N)]) :- int(N),!.

translate(X,[push(X)]) :- atom(X),!.

translate(add(A,B),Target) :-
    translate(A,TargetA),
    translate(B,TargetB),
    TargetOP = [add],
    flatten([TargetA,TargetB,TargetOP],Target),!.

translate(sub(A,B),Target) :-
    translate(A,TargetA),
    translate(B,TargetB),
    TargetOP = [sub],
    flatten([TargetA,TargetB,TargetOP],Target),!.

translate(mult(A,B),Target) :-
    translate(A,TargetA),
    translate(B,TargetB),
    TargetOP = [mult],
    flatten([TargetA,TargetB,TargetOP],Target),!.
```

Translational Semantics

```
?- translate(add(3,2),C),ppc(C).
    push(3)
    push(2)
    add
C = [push(3), push(2), add].

?- translate(add(3,mult(2,x)),C),ppc(C).
    push(3)
    push(2)
    push(x)
    mult
    add
C = [push(3), push(2), push(x), mult, add].
```

Translational Semantics

The boolean expressions can be defined in a similar manner:

```
translate(true,Target) :-
    Target = [push(true)],!.

translate(false,Target) :-
    Target = [push(false)],!.

translate(and(A,B),Target) :-
    translate(A,T1),
    translate(B,T2),
    T3 = [and],
    flatten([T1,T2,T3],Target),!.

translate(or(A,B),Target) :-
    translate(A,T1),
    translate(B,T2),
    T3 = [or],
    flatten([T1,T2,T3],Target),!.

translate(not(A),Target) :-
    translate(A,T1),
    T2 = [neg],
    flatten([T1,T2],Target),!.
```

Translational semantics of statements:

```
translate(skip,[]) :- !.

translate(seq(C1,C2),Target) :-
    translate(C1,T1),
    translate(C2,T2),
    flatten([T1,T2],Target),!.

translate(assign(X,A),Target) :-
    translate(A,T1),
    T2 = [pop(X)],
    flatten([T1,T2],Target),!.

translate(if(B,C0,C1),Target) :-
    translate(B,T1),
    T2 = [jmpf(iflabel1)],
    translate(C0,T3),
    T4 = [jmp(iflabel2),label(iflabel1)],
    translate(C1,T5),
    T6 = [label(iflabel2)],
    flatten([T1,T2,T3,T4,T5,T6],Target),!.

translate(whiledo(B,C),Target) :-
    T1 = [label(whilelabel1)],
    translate(B,T2),
    T3 = [jmpf(whilelabel2)],
    translate(C,T4),
    T5 = [jmp(whilelabel1),label(whilelabel2)],
    flatten([T1,T2,T3,T4,T5],Target),!.
```

Translational Semantics

```
?- translate(assign(x,1) seq assign(y,mult(2,x)),C),ppc(C).
    push(1)
    pop(x)
    push(2)
    push(x)
    mult
    pop(y)
C = [push(1), pop(x), push(2), push(x), mult, pop(y)].
```

Note: the predicate ppc simply prints out the list of assembly code instructions in a nice formatted way.

Translational Semantics

```
?- translate(if(le(2,3),assign(i,3),assign(i,4)),C),ppc(C).
    push(2)
    push(3)
    le
    jmpf(iflabel1)
    push(3)
    pop(i)
    jmp(iflabel2)
label(iflabel1)
    push(4)
    pop(i)
label(iflabel2)
C = [push(2), push(3), le, jmpf(iflabel1), ...]
```

Translational Semantics

```
?- translate(whiledo(le(x,3),assign(x,add(x,1))),C),ppc(C).
label(whilelabel1)
    push(x)
    push(3)
    le
    jmpf(whilelabel2)
    push(x)
    push(1)
    add
    pop(x)
    jmp(whilelabel1)
label(whilelabel2)
C = [label(whilelabel1), push(x), push(3), le, jmpf(whilelabel2), ...]
```

Translational Semantics

The reason why we have compilers:

```
assign(i,1) seq
assign(z,1) seq
whiledo(not(eq(i,n)),
   assign(i,add(i,1)) seq
   assign(z,mult(z,i)))
```

```
    push(1)
    pop(i)
    push(1)
    pop(z)
label(whilelabel1)
    push(i)
    push(n)
    eq
    neg
    jmpf(whilelabel2)
    push(i)
    push(1)
    add
    pop(i)
    push(z)
    push(i)
    mult
    pop(z)
    jmp(whilelabel1)
label(whilelabel2)
```