Semantics of Imperative Statements

Imperative Statements

- *Assignment:* most essential.
- *Iteration statements*: `while` loops, `for` loops, `do..while` loops.
- *Decision statements, procedures, recursion*: not exclusive to imperative programming.

- Execution model:
  - Based on a notion of *state* — simplistically: the tuple of all variables in the program
  - Assignment changes the state.
  - Computation is the sequence of states that the program goes through.

Case Study: Python

```
x = 60
y = 144
while x != y :
    if x < y :
        y = y - x
    else :
        x = x - y
```

Case Study: Python

Assignments

```
x = 60
y = 144
while x != y :
    if x < y :
        y = y - x
    else :
        x = x - y
```

Case Study: Python

While condition

```
x = 60
y = 144
while x != y :
    if x < y :
        y = y - x
    else :
        x = x - y
```

While loop

While body

Case Study: Python

```
x = 60
y = 144
while x != y :
    if x < y :
        y = y - x
    else :
        x = x - y
```

If condition

"Then" branch

"Else" branch

If statement

Case Study: Python

```
x = 60
y = 144
while x != y :
    if x < y :
        y = y - x
    else :
        x = x - y
```

One statement per line, indentation defines blocks

Semantics

- ◇ Mathematical description of the execution model of a language.
- ◇ Essentially a translation mechanism.
- ◇ Assumption: we know the description language.
- ◇ The semantics helps us learn the new language (for which the semantics is defined).
- ◇ Each construct of the new language must be described somehow.
- ◇ Generalization: any translation of the new language into an already known language can be construed as semantics.
- ◇ Compiler: falls into this category too.

Toy Language

```
<Stmt> ::= <Variable> '=' <Expr>
         | 'if' <boolexpr> 'then' '{' <Stmt> '}'
                           'else' '{' <Stmt> '}'
         | 'if' <boolexpr> 'then '{' <Stmt> '}'
         | 'while <boolexpr> 'do' '{' <Stmt> '}'
         | '{' <Stmt> '}'
         | <Stmt> ';' <Stmt>
         | <Stmt> ;

<boolexpr> ::= <expr> '<' <expr> | <expr> '=<' <expr>
             | <expr> '==' <expr> | <expr> '\=' <expr>
             | <expr> '>' <expr> | <expr> '>=' <expr>

<expr> ::= ...same as before...
```

Vanilla Assembly Language

- Each toy language skeleton translated into a VAL skeleton.
- Simple and straightforward approach, not the most efficient VAL code.
- We need a stack for evaluating expressions.
- New register: `esp` – the stack register
  - Initialized with 10000
  - Each push implemented as
    `esp-=4;*(int*)&M[esp]=Register;`
  - Each pop implemented as
    `Register=*(int*)&M[esp];esp+=4;`

Types of Semantics

- Operational semantics
  - Small step
  - Big step
- Denotational semantics
- Collecting semantics
- Axiomatic semantics
- Various combinations of the above

Semantics of Toy Language

- Expressed as reasoning rules
- The context is an *environment*
  - Mapping from variable names to addresses
  - Assume that it already contains all the variables in the programs
  - Later we learn how to build it dynamically
- We assume we can generate new labels on the fly
  - Later we see how we can implement this
- Each rule handles the right hand side of a production in the grammar

Semantics of Expressions

$$\frac{}{\mathcal{E} \vdash [\![K]\!] = \text{''}\texttt{esp -= 4 ; *(int*)\&M[esp] = K ;}\text{''}} \quad K \text{ is a constant}$$

$$\frac{}{\mathcal{E} \vdash [\![V]\!] = \text{''}\begin{array}{l}\texttt{ecx = *(int*)\&M[}\mathcal{E}(V)\texttt{] ;}\\ \texttt{esp -= 4 ; *(int*)\&M[esp] = ecx ;}\end{array}\text{''}} \quad V \text{ is a variable}$$

$$\frac{\mathcal{E} \vdash [\![E_1]\!] = C_1 \qquad \mathcal{E} \vdash [\![E_2]\!] = C_2}{\mathcal{E} \vdash [\![E_1 \oplus E2]\!] = \text{''}\begin{array}{l}C_1\\ C_2\\ \texttt{ecx = *(int*)\&M[esp] ; esp += 4 ;}\\ \texttt{eax = *(int*)\&M[esp] ; esp += 4 ;}\\ \texttt{eax }\oplus\texttt{= ecx ;}\\ \texttt{esp -= 4 ; *(int*)\&M[esp] = eax ;}\end{array}\text{''}} \quad \oplus \in \{+, -, *, /\}$$

Semantics of Assignments

$$\frac{\mathcal{E} \vdash [\![\, E \,]\!] = C}{\mathcal{E} \vdash [\![\, V = E \,]\!] = \text{''}\; C \;\; \texttt{ecx = *(int*)\&M[esp] ; esp += 4 ;} \;\; \texttt{*(int*)\&M[}\mathcal{E}(V)\texttt{] = ecx ;}\text{''}} \quad V \text{ is a variable}$$

◇ After execution of code C, which implements the expression E, the result is expected at the top of the stack.

◇ The current code pops the value from the stack, and transfers it into the memory location to which variable V is mapped to.

Semantics of "If" Statements

$$\frac{\mathcal{E} \vdash [\![\, E_1 \,]\!] = C_1 \qquad \mathcal{E} \vdash [\![\, E_2 \,]\!] = C_2 \qquad \mathcal{E} \vdash [\![\, S_1 \,]\!] = C_3 \qquad \mathcal{E} \vdash [\![\, S_2 \,]\!] = C_4}{\mathcal{E} \vdash [\![\texttt{if(}E_1 \oplus E_2\texttt{)then } S_1 \text{ else } S_2 \,]\!] = \begin{array}{ll} & C_1 \\ & C_2 \\ '' & \texttt{ecx = *(int*)\&M[esp] ; esp += 4 ;} \\ & \texttt{eax = *(int*)\&M[esp] ; esp += 4 ;} \\ & \texttt{if (eax } \oplus \texttt{ ecx) goto Lthen ;}'' \\ & C_4 \\ '' & \texttt{goto Lendif;} \\ & \texttt{Lthen:}'' \\ & C_3 \\ '' & \texttt{Lendif:}'' \end{array}} \quad \oplus \in \{<, >, =<, >=, ==, != \}$$

Semantics of "While" Statements

$$\frac{\mathcal{E} \vdash [\![\, E_1 \,]\!] = C_1 \qquad \mathcal{E} \vdash [\![\, E_2 \,]\!] = C_2 \qquad \mathcal{E} \vdash [\![\, S \,]\!] = C_3}{\mathcal{E} \vdash [\![\texttt{while(}E_1 \oplus E_2\texttt{)do } S\,]\!] = \begin{array}{ll} '' & \texttt{Lwhile:}\ '' \\ & C_1 \\ & C_2 \\ '' & \texttt{ecx = *(int*)\&M[esp] ; esp += 4 ;} \\ & \texttt{eax = *(int*)\&M[esp] ; esp += 4 ;} \\ & \texttt{if (eax } \oplus \texttt{ ecx) goto Lwhilebody ;} \\ & \texttt{goto Lendwhile;} \\ & \texttt{Lwhilebody:}\ '' \\ & C_3 \\ '' & \texttt{goto Lwhile;} \\ & \texttt{Lendwhile:}\ '' \end{array}} \quad \oplus \in \{<, >, =<, >=, ==, != \}$$

A Little Compiler: Expressions

```
compileExpr(K,E,E,T,T) :-
        integer(K),!,
        write('    esp -= 4 ; *(int*)&M[esp] = '),
        write(K),write(' ; // push '), writeln(K).

compileExpr(V,Ein,Eout,Tin,Tout) :-
        atom(V),!,
        (   member((V->Addr),Ein)
        ->  Tout = Tin, Eout = Ein
        ;   Tout is Tin+4, Eout = [(V->Tin)|Ein], Addr = Tin),
        write('    ecx = *(int*)&M['),
        write(Addr),
        write('] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push '),
        writeln(V).

compileExpr(Exp,Ein,Eout,Tin,Tout) :-
        Exp =.. [O,A,B],
        compileExpr(A,Ein,Eaux,Tin,Taux),
        compileExpr(B,Eaux,Eout,Taux,Tout),
        writeln('    ecx = *(int*)&M[esp] ; esp += 4 ;'),
        writeln('    eax = *(int*)&M[esp] ; esp += 4 ;'),
        write('    eax '), write(O), writeln('= ecx ;'),
        write('    esp -= 4 ; *(int*)&M[esp] = eax ; // push result of '),
        writeln(O).
```

Rule for Assignment

```
compile(V=E,Ein,Eout,Tin,Tout,L,L) :-
        compileExpr(E,Ein,Eaux,Tin,Taux),
        (   member((V->Addr),Eaux)
        ->  Tout = Taux, Eout = Eaux
        ;   Tout is Taux+4, Eout = [(V->Taux)|Eaux], Addr = Taux),
        writeln('    ecx = *(int*)&M[esp] ; esp += 4 ;'),
        write('    *(int*)&M['),write(Addr),
        write('] = ecx ; // pop '),
        writeln(V).
```

```
?- compile((x=y+2),[],Eout,0,Tout,_,_).
    ecx = *(int*)&M[0] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push y
    esp -= 4 ; *(int*)&M[esp] = 2 ; // push 2
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    eax += ecx ;
    esp -= 4 ; *(int*)&M[esp] = eax ; // push result of +
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[4] = ecx ; // pop x
Eout = [ (x->4), (y->0)],
Tout = 8.
```

Rule for "If" Statement

```
compile(if B then S1 else S2,Ein,Eout,Tin,Tout,Lin,Lout) :- !,
        B =.. [O,X,Y], La1 is Lin+1,
        (   O == (\=) -> Otrans = '!=' ; Otrans = O ),
        writeln('    // start of if-then-else statement'),
        compileExpr(X,Ein,Ea1,Tin,Ta1),
        compileExpr(Y,Ea1,Ea2,Ta1,Ta2),
        writeln('    ecx = *(int*)&M[esp] ; esp += 4 ;') ,
        writeln('    eax = *(int*)&M[esp] ; esp += 4 ;') ,
        write('    if ( eax '), write(Otrans),
        write(' ecx ) goto Lthen'),
        write(Lin), writeln('; // if condition'),
        compile(S2,Ea2,Ea3,Ta2,Ta3,La1,La2),
        write('    goto Lendif'),write(Lin),writeln(';'),
        write('Lthen'),write(Lin),writeln(':'),
        compile(S1,Ea3,Eout,Ta3,Tout,La2,Lout),
        write('Lendif'),write(Lin),writeln(':').
```

"If" Example

```
?- compile(if x < y then a = 1 else b = 2,[],Eout,0,Tout,0,_).
    // start of if-then-else statement
    ecx = *(int*)&M[0] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push x
    ecx = *(int*)&M[4] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push y
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    if ( eax < ecx ) goto Lthen0; // if condition
    esp -= 4 ; *(int*)&M[esp] = 2 ; // push 2
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[8] = ecx ; // pop b
    goto Lendif0;
Lthen0:
    esp -= 4 ; *(int*)&M[esp] = 1 ; // push 1
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[12] = ecx ; // pop a
Lendif0:
Eout = [ (a->12), (b->8), (y->4), (x->0)],
Tout = 16.
```

Rule for "While"

```
compile(while B do S,Ein,Eout,Tin,Tout,Lin,Lout) :- !,
        B =.. [O,X,Y], La1 is Lin+1,
        (   O == (\=) -> Otrans = '!=' ; Otrans = O ),
        write('Lwhile'),write(Lin),writeln(':'),
        compileExpr(X,Ein,Ea1,Tin,Ta1),
        compileExpr(Y,Ea1,Ea2,Ta1,Ta2),
        writeln('    ecx = *(int*)&M[esp] ; esp += 4 ;') ,
        writeln('    eax = *(int*)&M[esp] ; esp += 4 ;') ,
        write('    if ( eax '), write(Otrans),
        write(' ecx ) goto Lwhilebody'), write(Lin), writeln(';'),
        write('    goto Lendwhile'),write(Lin),writeln(';'),
        write('Lwhilebody'),write(Lin),writeln(':'),
        compile(S,Ea2,Eout,Ta2,Tout,La1,Lout),
        write('    goto Lwhile'),write(Lin),writeln(';'),
        write('Lendwhile'),write(Lin),writeln(':').
```

"While" Example

```
?- compile(while x < y do x=x+1,[],Eout,0,Tout,0,_).
Lwhile0:
    ecx = *(int*)&M[0] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push x
    ecx = *(int*)&M[4] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push y
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    if ( eax < ecx ) goto Lwhilebody0;
    goto Lendwhile0;
Lwhilebody0:
    ecx = *(int*)&M[0] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push x
    esp -= 4 ; *(int*)&M[esp] = 1 ; // push 1
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    eax += ecx ;
    esp -= 4 ; *(int*)&M[esp] = eax ; // push result of +
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[0] = ecx ; // pop x
    goto Lwhile0;
Lendwhile0:
Eout = [ (y->4), (x->0)],
Tout = 8.
```

The Rest of the Rules

```
compile(S1;S2,Ein,Eout,Tin,Tout,Lin,Lout) :- !,
        compile(S1,Ein,Eaux,Tin,Taux,Lin,Laux),
        compile(S2,Eaux,Eout,Taux,Tout,Laux,Lout).

compile(S;,Ein,Eout,Tin,Tout,Lin,Lout) :- !,
        compile(S,Ein,Eout,Tin,Tout,Lin,Lout).

compile({S},Ein,Eout,Tin,Tout,Lin,Lout) :- !,
        compile(S,Ein,Eout,Tin,Tout,Lin,Lout).
```

Generating a Full C Program

```
compileProg(P) :-
        writeln('#include <stdio.h>'),
        writeln('int eax,ebx,ecx,edx,esi,edi,ebp,esp;'),
        writeln('unsigned char M[10000];'),
        writeln('void exec(void) {'),
        compile(P,[],Eout,0,_,0,_),
        writeln('{}}'),nl,
        writeln('int main() {'),
        writeln('    esp = 10000 ;'),
        writeln('    exec();'),
        outputVars(Eout),
        writeln('    return 0;'),
        writeln('}').

outputVars([]).
outputVars([(V->Addr)|T]) :-
        write('    printf("'),write(V),write('=%d\\n",'),
        write('*(int*)&M['),write(Addr),writeln(']);'),
        outputVars(T).
```

A Full-Fledged Example

```
?- P = (
          x = 144 ;
          y = 60 ;
          while ( x \= y ) do {
             if ( x < y ) then {
                y = y - x ;
             } else {
                x = x - y ;
             } ;
          } ;
       ),
         compileProg(P).
```

Resulting C Program

```
#include <stdio.h>
int eax,ebx,ecx,edx,esi,edi,ebp,esp;
unsigned char M[10000];
void exec(void) {
    esp -= 4 ; *(int*)&M[esp] = 144 ; // push 144
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[0] = ecx ; // pop x
    esp -= 4 ; *(int*)&M[esp] = 60 ; // push 60
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[4] = ecx ; // pop y
Lwhile0:
    ecx = *(int*)&M[0] ; esp -= 4 ;
    *(int*)&M[esp] = ecx ; // push x
    ecx = *(int*)&M[4] ; esp -= 4 ;
    *(int*)&M[esp] = ecx ; // push y
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    if ( eax != ecx ) goto Lwhilebody0;
    goto Lendwhile0;
Lwhilebody0:
    // start of if-then-else statement
    ecx = *(int*)&M[0] ; esp -= 4 ;
    *(int*)&M[esp] = ecx ; // push x
    ecx = *(int*)&M[4] ; esp -= 4 ;
    *(int*)&M[esp] = ecx ; // push y
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    if ( eax < ecx ) goto Lthen1;
```

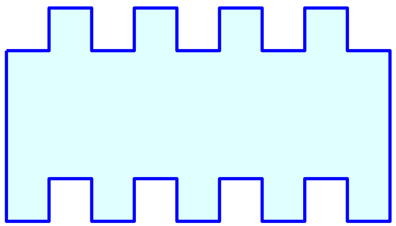
```
    ecx = *(int*)&M[0] ; esp -= 4 ;
    *(int*)&M[esp] = ecx ; // push x
    ecx = *(int*)&M[4] ; esp -= 4 ;
    *(int*)&M[esp] = ecx ; // push y
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    eax -= ecx ;
    esp -= 4 ;
   *(int*)&M[esp] = eax ; // push result of -
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[0] = ecx ; // pop x
    goto Lendif1;
Lthen1:
    ecx = *(int*)&M[4] ; esp -= 4 ;
    *(int*)&M[esp] = ecx ; // push y
    ecx = *(int*)&M[0] ; esp -= 4 ;
    *(int*)&M[esp] = ecx ; // push x
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    eax -= ecx ; esp -= 4 ;
    *(int*)&M[esp] = eax ; // push result of -
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[4] = ecx ; // pop y
Lendif1:
    goto Lwhile0;
Lendwhile0: {}}

int main() {
    esp = 10000 ;
    exec();
    printf("y=%d\n",*(int*)&M[4]);
    printf("x=%d\n",*(int*)&M[0]);
    return 0;
}
```

Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture

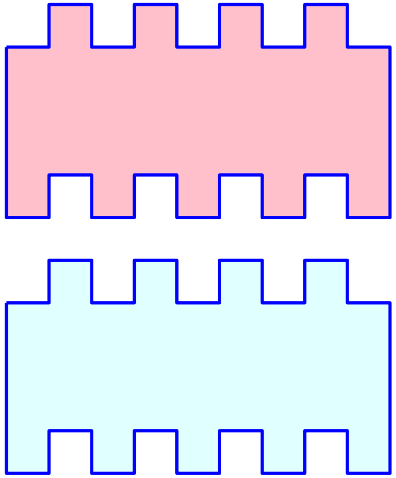
Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



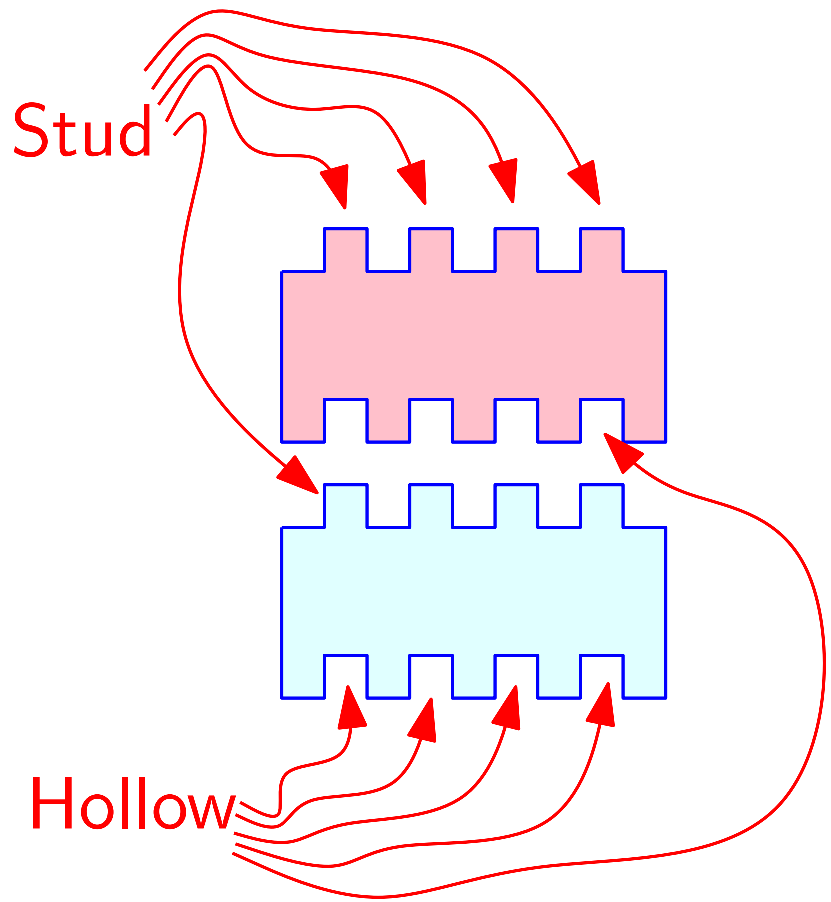
Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



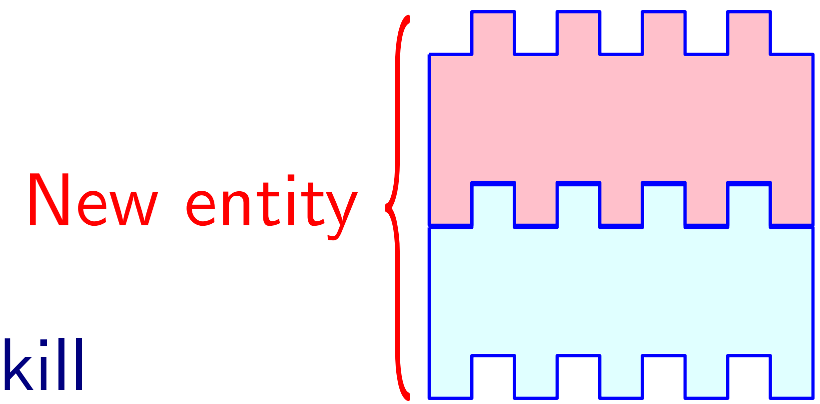
Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



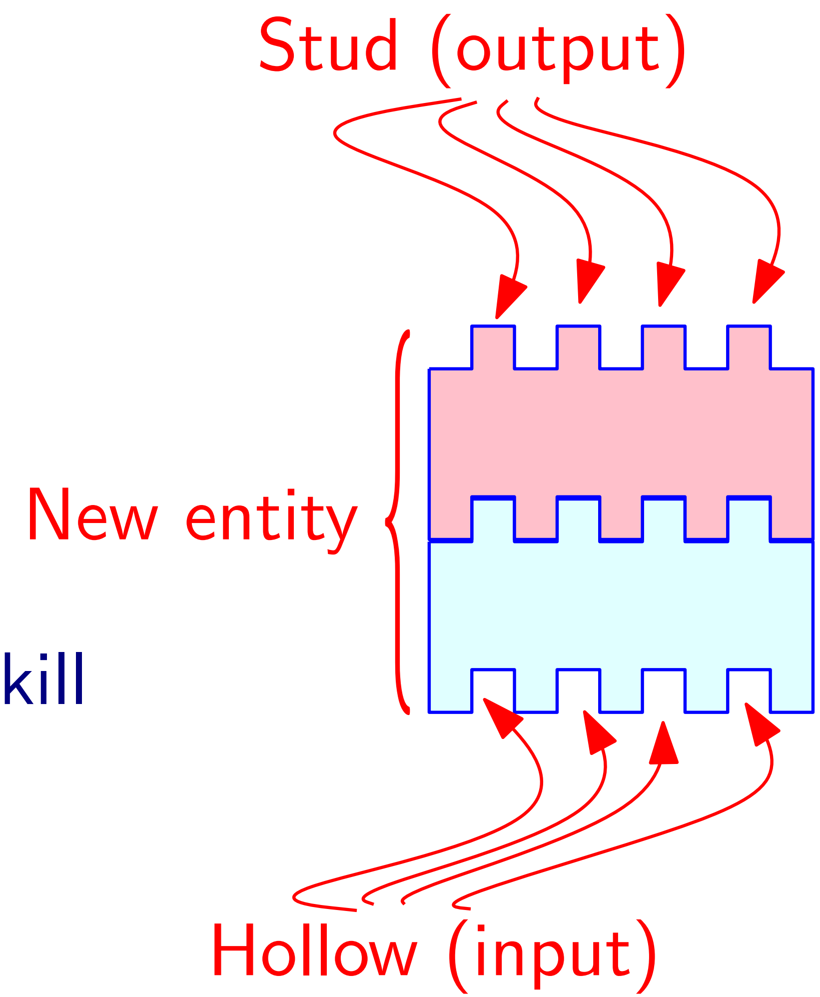
Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



The new entitiy can be further combined!

What Have We Learned

- Semantics is a description of the execution model in a language we supposedly already know
- Compilation is an instance of giving a semantics to a language
- *Compositionality:* important principle that enables the process of defining a semantics
- Prolog is perfectly equipped for writing toy compilers by following semantic rules.