CMSC 330: Organization of Programming Languages

Operational Semantics
Formal Semantics of a Prog. Lang.

- Mathematical description of the meaning of programs written in that language
  - What a program computes, and what it does

- Three main approaches to formal semantics
  - Denotational
  - Operational
  - Axiomatic
This Course: Operational Semantics

- We will show how an operational semantics may be defined for Micro-Ocaml
  - And develop an interpreter for it, along the way

- Approach: use rules to define a judgment

\[ e \Rightarrow v \]

- Says “\( e \) evaluates to \( v \)”
- \( e \): expression in Micro-OCaml
- \( v \): value that results from evaluating \( e \)
Definitional Interpreter

- It turns out that the rules for judgment \( e \Rightarrow v \) can be easily turned into idiomatic OCaml code
  - The language’s expressions \( e \) and values \( v \) have corresponding OCaml datatype representations \( \text{exp} \) and \( \text{value} \)
  - The semantics is represented as a function
    
    \[
    \text{eval}: \ \text{exp} \rightarrow \text{value}
    \]

- This way of presenting the semantics is referred to as a definitional interpreter
  - The interpreter defines the language’s meaning
Micro-OCaml Expression Grammar

e ::= x | n | e + e | let x = e in e

- e, x, n are **meta-variables** that stand for categories of syntax
  - x is any identifier (like z, y, foo)
  - n is any numeral (like 1, 0, 10, -25)
  - e is any expression (here defined, recursively!)

- Concrete syntax of actual expressions in **black**
  - Such as let, +, z, foo, in, ...

  ::= and | are **meta-syntax** used to define the syntax of a language (part of “Backus-Naur form,” or BNF)
Micro-OCaml Expression Grammar

\[ e ::= x | n | e + e | \text{let } x = e \text{ in } e \]

Examples

- 1 is a numeral \( n \) which is an expression \( e \)
- 1+z is an expression \( e \) because
  - 1 is an expression \( e \),
  - \( z \) is an identifier \( x \), which is an expression \( e \), and
  - \( e + e \) is an expression \( e \)
- let \( z = 1 \) in \( 1+z \) is an expression \( e \) because
  - \( z \) is an identifier \( x \),
  - 1 is an expression \( e \),
  - 1+z is an expression \( e \), and
  - let \( x = e \) in \( e \) is an expression \( e \)
Abstract Syntax = Structure

- Here, the grammar for $e$ is describing its abstract syntax tree (AST), i.e., $e$’s structure

$$e ::= x \mid n \mid e + e \mid \text{let } x = e \text{ in } e$$

corresponds to (in definitional interpreter)

```plaintext
type id = string
type num = int
type exp =
    | Ident of id (* x *)
    | Num of num (* n *)
    | Plus of exp * exp (* e+e *)
    | Let of id * exp * exp
        (* let x=e in e * )
```
Aside: Real Interpreters

Parser

Optional Static Analyzer (e.g., Type Checker)

Abstract Syntax Tree (AST), a kind of intermediate representation (IR)

Evaluator
the part we write in the definitional interpreter

Source → Front End → Back End → Output

Input
Values

- An expression’s final result is a value. What can values be?
  \[ v ::= n \]

- Just numerals for now
  - In terms of an interpreter’s representation:
    \[ \text{type value = int} \]
  - In a full language, values \( v \) will also include booleans (true, false), strings, functions, ...
Defining the Semantics

- Use rules to define judgment $e \Rightarrow v$

- Judgments are just statements. We use rules to prove that the statement is true.
  - $1+3 \Rightarrow 4$
    - $1+3$ is an expression $e$, and $4$ is a value $v$
    - This judgment claims that $1+3$ evaluates to $4$
    - We use rules to prove it to be true
  - let foo=1+2 in foo+5 $\Rightarrow 8$
  - let f=1+2 in let z=1 in f+z $\Rightarrow 4$
Suppose $e$ is a numeral $n$
  • Then $e$ evaluates to itself, i.e., $n \Rightarrow n$

Suppose $e$ is an addition expression $e_1 + e_2$
  • If $e_1$ evaluates to $n_1$, i.e., $e_1 \Rightarrow n_1$
  • If $e_2$ evaluates to $n_2$, i.e., $e_2 \Rightarrow n_2$
  • Then $e$ evaluates to $n_3$, where $n_3$ is the sum of $n_1$ and $n_2$
  • I.e., $e_1 + e_2 \Rightarrow n_3$

Suppose $e$ is a let expression $\text{let } x = e_1 \text{ in } e_2$
  • If $e_1$ evaluates to $v$, i.e., $e_1 \Rightarrow v_1$
  • If $e_2\{v_1/x\}$ evaluates to $v_2$, i.e., $e_2\{v_1/x\} \Rightarrow v_2$
    • Here, $e_2\{v_1/x\}$ means “the expression after substituting occurrences of $x$ in $e_2$ with $v_1$”
  • Then $e$ evaluates to $v_2$, i.e., $\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2$
Rules of Inference

- We can use a more compact notation for the rules we just presented: rules of inference
  - Has the following format
    \[
    \begin{array}{c}
    H_1 \quad \ldots \quad H_n \\
    \hline
    C
    \end{array}
    \]
  - Says: if the conditions \( H_1 \ldots H_n \) ("hypotheses") are true, then the condition \( C \) ("conclusion") is true
  - If \( n=0 \) (no hypotheses) then the conclusion automatically holds; this is called an axiom

- We are using inference rules where \( C \) is our judgment about evaluation, i.e., that \( e \Rightarrow v \)
Lego Blocks and Lego Cars

P = 8.0 mm
= 5/6 × H
= 2.5 × h

h = 3.2 mm
= 1/3 × H
= 0.4 × P

2 × P = 0.2 mm
= 15.8 mm

H = 9.6 mm
= 3 × h
= 1.2 × P

P − 0.2 mm
= 7.8 mm
Rules of Inference: Num and Sum

- Suppose $e$ is a numeral $n$
  - Then $e$ evaluates to itself, i.e., $n \Rightarrow n$

- Suppose $e$ is an addition expression $e_1 + e_2$
  - If $e_1$ evaluates to $n_1$, i.e., $e_1 \Rightarrow n_1$
  - If $e_2$ evaluates to $n_2$, i.e., $e_2 \Rightarrow n_2$
  - Then $e$ evaluates to $n_3$, where $n_3$ is the sum of $n_1$ and $n_2$
  - I.e., $e_1 + e_2 \Rightarrow n_3$
Rules of Inference: Let

- Suppose $e$ is a let expression $\text{let } x = e_1 \text{ in } e_2$
  - If $e_1$ evaluates to $v$, i.e., $e_1 \Rightarrow v_1$
  - If $e_2\{v_1/x\}$ evaluates to $v_2$, i.e., $e_2\{v_1/x\} \Rightarrow v_2$
  - Then $e$ evaluates to $v_2$, i.e., $\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2$

<table>
<thead>
<tr>
<th>$e_1 \Rightarrow v_1$</th>
<th>$e_2{v_1/x} \Rightarrow v_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>let</strong> $x = e_1 \text{ in } e_2 \Rightarrow v_2$</td>
<td></td>
</tr>
</tbody>
</table>
Derivations

- When we apply rules to an expression in succession, we produce a derivation
  - It’s a kind of tree, rooted at the conclusion

- Produce a derivation by goal-directed search
  - Pick a rule that could prove the goal
  - Then repeatedly apply rules on the corresponding hypotheses

  ➢ Goal: Show that \( \text{let } x = 4 \text{ in } x+3 \Rightarrow 7 \)
## Derivations

### Goal: show that

\[
\text{let } x = 4 \text{ in } x + 3 \Rightarrow 7
\]

<table>
<thead>
<tr>
<th>(n \Rightarrow n)</th>
<th>(e_1 \Rightarrow n_1)</th>
<th>(e_2 \Rightarrow n_2)</th>
<th>(n_3 \text{ is } n_1 + n_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e_1 + e_2 \Rightarrow n_3)</td>
<td>(e_1 \Rightarrow v_1)</td>
<td>(e_2{v_1/x} \Rightarrow v_2)</td>
<td>(\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
4 & \Rightarrow 4 \\
3 & \Rightarrow 3 \\
7 & \text{ is } 4 + 3
\end{align*}
\]

\[
\begin{align*}
4 & \Rightarrow 4 \\
4 + 3 & \Rightarrow 7
\end{align*}
\]

\[
\text{let } x = 4 \text{ in } x + 3 \Rightarrow 7
\]
Quiz 1

What is derivation of the following judgment?

\[ 2 + (3 + 8) \Rightarrow 13 \]

(a)

\[
\begin{align*}
2 & \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

(b)

\[
\begin{align*}
3 & \Rightarrow 3 \\
8 & \Rightarrow 8 \\
\hline
3 + 8 & \Rightarrow 11 \\
2 & \Rightarrow 2 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

(c)

\[
\begin{align*}
8 & \Rightarrow 8 \\
3 & \Rightarrow 3 \\
11 & \text{is } 3+8 \\
\hline
2 & \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
13 & \text{is } 2+11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]
Quiz 1

What is derivation of the following judgment?

2 + (3 + 8) ⇒ 13

(a)

2 ⇒ 2
3 + 8 ⇒ 11

---------------------

2 + (3 + 8) ⇒ 13

(b)

3 ⇒ 3
8 ⇒ 8

------------

3 + 8 ⇒ 11
2 ⇒ 2

---------------------

2 + (3 + 8) ⇒ 13

(c)

8 ⇒ 8
3 ⇒ 3
11 is 3+8

------------

2 ⇒ 2
3 + 8 ⇒ 11
13 is 2+11

---------------------

2 + (3 + 8) ⇒ 13
The style of rules lends itself directly to the implementation of an interpreter as a recursive function.

```ocaml
let rec eval (e:exp):value =  
  match e with  
  | Ident x -> (* no rule *)  
    failwith "no value"  
  | Num n -> n  
  | Plus (e1,e2) ->  
    let n1 = eval e1 in  
    let n2 = eval e2 in  
    let n3 = n1+n2 in  
    n3  
  | Let (x,e1,e2) ->  
    let v1 = eval e1 in  
    let e2' = subst v1 x e2 in  
    let v2 = eval e2' in v2
```

Trace of evaluation of `eval` function corresponds to a derivation by the rules.
Derivations = Interpreter Call Trees

\[
\begin{align*}
4 \Rightarrow 4 & \quad 3 \Rightarrow 3 & \quad 7 \text{ is } 4+3 \\

4 \Rightarrow 4 & \quad 4+3 \Rightarrow 7 \\
\text{let } x = 4 \text{ in } x+3 \Rightarrow 7
\end{align*}
\]

Has the same shape as the recursive call tree of the interpreter:

\[
\begin{align*}
\text{eval } \text{Num } 4 \Rightarrow 4 & \quad \text{eval } \text{Num } 3 \Rightarrow 3 & \quad 7 \text{ is } 4+3 \\

\text{eval } \left( \text{subst } 4 \ "x" \right) \\
\text{eval } \text{Num } 4 \Rightarrow 4 & \quad \text{Plus}(..., \text{Num } 3) \Rightarrow 7 \\
\text{eval } \text{Let}(..., \text{Num } 4, \text{Plus}(..., \text{Num } 3)) \Rightarrow 7
\end{align*}
\]
Semantics Defines Program Meaning

- $e \Rightarrow v$ holds if and only if a proof can be built
  - Proofs are derivations: axioms at the top, then rules whose hypotheses have been proved to the bottom
  - No proof means $e \not\Rightarrow v$
- Proofs can be constructed bottom-up
  - In a goal-directed fashion
- Thus, function $\text{eval} \ e = \{ v \mid e \Rightarrow v \}$
  - Determinism of semantics implies at most one element for any $e$
- So: Expression $e$ means $v$
Environment-style Semantics

- The previous semantics uses substitution to handle variables
  - As we evaluate, we replace all occurrences of a variable \( x \) with values it is bound to

- An alternative semantics, closer to a real implementation, is to use an environment
  - As we evaluate, we maintain an explicit map from variables to values, and look up variables as we see them
Mathematically, an environment is a partial function from identifiers to values

- If $A$ is an environment, and $x$ is an identifier, then $A(x)$ can either be …
- … a value (intuition: the variable has been declared)
- … or undefined (intuition: variable has not been declared)

An environment can also be thought of as a table

- If $A$ is

<table>
<thead>
<tr>
<th>Id</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>0</td>
</tr>
<tr>
<td>$y$</td>
<td>2</td>
</tr>
</tbody>
</table>

- then $A(x)$ is 0, $A(y)$ is 2, and $A(z)$ is undefined
Notation, Operations on Environments

• is the empty environment (undefined for all ids)

If $A$ is an environment then $A, x: v$ is one that extends $A$ with a mapping from $x$ to $v$

• Sometimes just write $x: v$ instead of $\cdot, x: v$ for brevity

• NB. if $A$ maps $x$ to some $v'$, then that mapping is shadowed by the mapping $x: v$

Lookup $A(x)$ is defined as follows

$\cdot(x) = \text{undefined}$

$$
(A, y: v)(x) = \begin{cases} 
v & \text{if } x = y \\
A(x) & \text{if } x <> y \text{ and } A(x) \text{ defined} \\
\text{undefined} & \text{otherwise}
\end{cases}
$$
An environment is just a list of mappings, which are just pairs of variable to value - called an association list
Semantics with Environments

- The environment semantics changes the judgment
  \[ e \Rightarrow v \]
  to be
  \[ A; e \Rightarrow v \]
  where \( A \) is an environment
  - Idea: \( A \) is used to give values to the identifiers in \( e \)
  - \( A \) can be thought of as containing declarations made up to \( e \)

- Previous rules can be modified by
  - Inserting \( A \) everywhere in the judgments
  - Adding a rule to look up variables \( x \) in \( A \)
  - Modifying the rule for `let` to add \( x \) to \( A \)
### Environment-style Rules

1. **Look up variable** \( x \) **in environment** \( A \):
   
   \[
   A(x) = v \\
   \overline{A; x \Rightarrow v}
   \]

2. **Extend environment** \( A \) **with mapping from** \( x \) **to** \( v \):
   
   \[
   A; e1 \Rightarrow v1 \\
   A,x:v1; e2 \Rightarrow v2 \\
   \overline{A; let \ x = e1 \ in \ e2 \Rightarrow v2}
   \]

3. **Add two numbers** \( n1 \) **and** \( n2 \) **to get** \( n3 \):
   
   \[
   A; e1 \Rightarrow n1 \\
   A; e2 \Rightarrow n2 \\
   \overline{n3 \text{ is } n1+n2} \\
   A; e1 + e2 \Rightarrow n3
   \]
let rec eval env e =
    match e with
    | Ident x -> lookup env x
    | Num n -> n
    | Plus (e1,e2) ->
        let n1 = eval env e1 in
        let n2 = eval env e2 in
        let n3 = n1+n2 in
        n3
    | Let (x,e1,e2) ->
        let v1 = eval env e1 in
        let env' = extend env x v1 in
        let v2 = eval env' e2 in v2
Quiz 2

What is a derivation of the following judgment?

•; let x=3 in x+2 ⇒ 5

(a)  
\[ \begin{align*} 
  & x \Rightarrow 3 \\
  & 2 \Rightarrow 2 \\
  & 5 \text{ is } 3+2 \\
\end{align*} \]

\[ \begin{align*} 
  & 3 \Rightarrow 3 \\
  & x+2 \Rightarrow 5 \\
\end{align*} \]

\[ \begin{align*} 
  & \text{let x=3 in x+2 ⇒ 5} \\
\end{align*} \]

(b)  
\[ \begin{align*} 
  & x:3; x \Rightarrow 3 \\
  & x:3; 2 \Rightarrow 2 \\
  & 5 \text{ is } 3+2 \\
\end{align*} \]

\[ \begin{align*} 
  & 3 \Rightarrow 3 \\
  & x:3; x+2 \Rightarrow 5 \\
\end{align*} \]

\[ \begin{align*} 
  & \text{let x=3 in x+2 ⇒ 5} \\
\end{align*} \]

(c)  
\[ \begin{align*} 
  & x:2; x \Rightarrow 3 \\
  & x:2; 2 \Rightarrow 2 \\
  & 5 \text{ is } 3+2 \\
\end{align*} \]

\[ \begin{align*} 
  & \text{let x=3 in x+2 ⇒ 5} \\
\end{align*} \]
Quiz 2

What is a derivation of the following judgment?

\( \cdot; \text{let } x=3 \text{ in } x+2 \Rightarrow 5 \)

(a)

\[
\begin{align*}
\text{x} & \Rightarrow 3 \\
2 & \Rightarrow 2 \\
5 & \text{is } 3+2 \\
\hline
3 & \Rightarrow 3 \\
x+2 & \Rightarrow 5 \\
\hline
\text{let } x=3 \text{ in } x+2 & \Rightarrow 5
\end{align*}
\]

(b)

\[
\begin{align*}
\text{x:3}; \text{x} & \Rightarrow 3 \\
x:3; 2 & \Rightarrow 2 \\
5 & \text{is } 3+2 \\
\hline
\cdot; 3 & \Rightarrow 3 \\
x:3; \text{x+2} & \Rightarrow 5 \\
\hline
\cdot; \text{let } x=3 \text{ in } x+2 & \Rightarrow 5
\end{align*}
\]

(c)

\[
\begin{align*}
\text{x:2}; \text{x} & \Rightarrow 3 \\
x:2; 2 & \Rightarrow 2 \\
5 & \text{is } 3+2 \\
\hline
\cdot; \text{let } x=3 \text{ in } x+2 & \Rightarrow 5
\end{align*}
\]
Adding Conditionals to Micro-OCaml

\[ e ::= x | v | e + e | \text{let } x = e \text{ in } e \]
\[ \quad | \text{eq0 } e | \text{if } e \text{ then } e \text{ else } e \]

\[ v ::= n | \text{true} | \text{false} \]

• In terms of interpreter definitions:

```ocaml
type exp =
| \text{Val of value}
| ... (* as before *)
| \text{Eq0 of } exp
| \text{If of } exp * exp * exp

type value =
| \text{Int of int}
| \text{Bool of bool}
```

"CMSC 330 Fall 2020""
Rules for Eq0 and Booleans

- Booleans evaluate to themselves
  - \( A; \text{false} \Rightarrow \text{false} \)

- eq0 tests for 0
  - \( A; \text{eq0 0} \Rightarrow \text{true} \)
  - \( A; \text{eq0 3+4} \Rightarrow \text{false} \)
Rules for Conditionals

- \(A; \text{e1} \Rightarrow \text{true} \quad A; \text{e2} \Rightarrow v\)
- \(A; \text{if e1 then e2 else e3} \Rightarrow v\)
- \(A; \text{e1} \Rightarrow \text{false} \quad A; \text{e3} \Rightarrow v\)
- \(A; \text{if e1 then e2 else e3} \Rightarrow v\)

- Notice that only one branch is evaluated
  - \(A; \text{if eq0 0 then 3 else 4} \Rightarrow 3\)
  - \(A; \text{if eq0 1 then 3 else 4} \Rightarrow 4\)
Quiz 3

What is the derivation of the following judgment?

•; if eq0 3-2 then 5 else 10 ⇒ 10

(a)
•; 3 ⇒ 3   •; 2 ⇒ 2   3-2 is 1
-------------------
•; eq0 3-2 ⇒ false   •; 10 ⇒ 10
-------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10

(b)
3 ⇒ 3   2 ⇒ 2
3-2 is 1
-------------------
eq0 3-2 ⇒ false   10 ⇒ 10
-------------------
if eq0 3-2 then 5 else 10 ⇒ 10

(c)
•; 3 ⇒ 3
•; 2 ⇒ 2
3-2 is 1
-------------------
•; eq0 3-2 ⇒ false   •; 10 ⇒ 10
-------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10

3-2 is 1 ≠ 0
-------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10
Quiz 3

What is the derivation of the following judgment?
•; if eq0 3-2 then 5 else 10 ⇒ 10

(a)
•; 3 ⇒ 3  •; 2 ⇒ 2  3-2 is 1
-----------------------------
•; eq0 3-2 ⇒ false  •; 10 ⇒ 10
-----------------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10

(b)
3 ⇒ 3  2 ⇒ 2
3-2 is 1
-------------
eq0 3-2 ⇒ false  10 ⇒ 10
-------------
if eq0 3-2 then 5 else 10 ⇒ 10

(c)
•; 3 ⇒ 3
•; 2 ⇒ 2
3-2 is 1
-----------------------------
•; 3-2 ⇒ 1  1 ≠ 0
-----------------------------
•; eq0 3-2 ⇒ false  •; 10 ⇒ 10
-----------------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10
Updating the Interpreter

```ocaml
let rec eval env e = 
    match e with
    | Ident x -> lookup env x
    | Val v -> v
    | Plus (e1,e2) -> 
        let Int n1 = eval env e1 in
        let Int n2 = eval env e2 in
        let n3 = n1+n2 in
        Int n3
    | Let (x,e1,e2) -> 
        let v1 = eval env e1 in
        let env' = extend env x v1 in
        let v2 = eval env' e2 in v2
    | Eq0 e1 -> 
        let Int n = eval env e1 in
        if n=0 then Bool true else Bool false
    | If (e1,e2,e3) -> 
        let Bool b = eval env e1 in
        if b then eval env e2
        else eval env e3
```

Basically both rules for `eq0` in this one snippet

Both `if` rules here
Quick Look: Type Checking

- Inference rules can also be used to specify a program’s **static semantics**
  - I.e., the rules for type checking
- We won’t cover this in depth in this course, but here is a flavor.

- **Types** $t ::= \text{bool} \mid \text{int}$
- **Judgment** $\vdash e : t$ says $e$ has type $t$
  - We define inference rules for this judgment, just as with the operational semantics
Some Type Checking Rules

- Boolean constants have type bool
  
  \[ \vdash \text{true} : \text{bool} \]
  
  \[ \vdash \text{false} : \text{bool} \]

- Equality checking has type bool too
  
  - Assuming its target expression has type int

  \[ \vdash e : \text{int} \]
  
  \[ \vdash \text{eq0} \ e : \text{bool} \]

- Conditionals

  \[ \vdash e1 : \text{bool} \]
  
  \[ \vdash e2 : t \]
  
  \[ \vdash e3 : t \]
  
  \[ \vdash \text{if} \ e1 \ \text{then} \ e2 \ \text{else} \ e3 : t \]
Handling Binding

- What about the types of variables?
  - Taking inspiration from the environment-style operational semantics, what could you do?

- Change judgment to be \( G \vdash e : t \) which says \( e \) has type \( t \) under type environment \( G \)
  - \( G \) is a map from variables \( x \) to types \( t \)
    - Analogous to map \( A \), but maps vars to types, not values

- What would be the rules for \texttt{let}, and variables?
Type Checking with Binding

- Variable lookup

\[ G(x) = t \]
\[ G \vdash x : t \]

- Let binding

\[ G \vdash e_1 : t_1 \quad G, x : t_1 \vdash e_2 : t_2 \]
\[ G \vdash \text{let } x = e_1 \text{ in } e_2 : t_2 \]

analogous to

\[ A(x) = v \]
\[ A; \ x \Rightarrow v \]

\[ A; e_1 \Rightarrow v_1 \quad A, x : v_1; e_2 \Rightarrow v_2 \]
\[ A; \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2 \]
Scaling up

Operational semantics (and similarly styled typing rules) can handle full languages
- With records, recursive variant types, objects, first-class functions, and more

Provides a concise notation for explaining what a language does. Clearly shows:
- Evaluation order
- Call-by-value vs. call-by-name
- Static scoping vs. dynamic scoping
- ... We may look at more of these later
Scaling Up: Lego City