Textual Reduction

CIS352 — Fall 2022
Kris Micinski
How does the computer evaluate this expression?

\[
(* (+ 2 (* 4 6)) (+ 3 5 7))
\]
A C-like language would **compile** the expression

```c
int x = (2 + 4*6) * (3 + 5 + 7);
```
Computer executes **instructions** on a **clock**
High-level observation:

ev\text{\textit{ery computation, in any language (running on your processor) is broken down—somehow—into sequences of atomic steps reified as instructions by your processor}}}
A key idea in the course is that evaluation of programs is often broken down into a sequence of small atomic steps.
A key idea in the course is that evaluation of programs is often broken down into a sequence of small atomic steps.

Assembly languages (from your systems course) are a special case where the processor’s execution makes each instruction atomic.
A key idea in the course is that evaluation of programs is often broken down into a sequence of small atomic steps.

Assembly languages (from your systems course) are a special case where the processor’s execution makes each *instruction* atomic.

Modern microprocessors involve lots of places where atomicity breaks down (cache coherence, etc..) but this is a key abstraction layer in computing.
In **high-level languages**, computations/expressions do **not** have one-to-one correspondence with the processor’s execution.

In fact, it is **impossible** (in general) to look at an expression and say how many steps the processor will take to execute an expression.

```plaintext
;;; Some number of steps
(* (+ 2 (* 4 6))
  (+ 3 5 7))
```
Textual reduction is a way of defining the semantics (i.e., meaning) of a program as a series of **progressing steps**, where each step consists of a program (represented **textually**), and a program to which it is “rewritten” (**textually reduced**).

Textual reduction semantics may be defined formally, but in this lecture we will be illustrating them informally.
This subexpression is reduced to...

\[
(+ (* 3 2) 1)
\]

\[
\rightarrow (+ 5 1)
\]

This expression, which is a value
Values

We often refer to the values of a programming language. Intuitively, a value is something that does not require any additional computation to manifest.

(+ 3 (* (foo 5) 6)) ;; not a value
’hello ;; value
15.0 ;; value
In terms of the computation, values are places where computation stops

(* 3 (+ 4 5))

In terms of our intuitive semantics: a builtin function may be applied when each of its arguments is a value
As an aside…
Later, we will see that this construction is inefficient: it means we are doing at least $O(n)$ work to (a) identify the redex and (b) then perform a transformation to obtain our result.
Later in the course we will see several improvements to this strategy, e.g., context-and-redex semantics or continuations.

At each point in time, we follow a two-step process: identify what can be reduced, and then perform the appropriate reduction.
Example reduction sequence

(* (* 3 1) (+ 4 5))
→ (* 3 (+ 4 5))
→ (* 3 9)
→ 27 ;; Resulting value
Question: in the last slide, why not do this?

\[
(* (* 3 1) (+ 4 5))
\rightarrow (* (* 3 1) 9)
\rightarrow (* 3 9)
\rightarrow 27 \; ;; \text{Resulting value}
\]

Answer: we could have! But typically we additionally constrain the reductions so that they occur in some predictable order.

In most PLs, we process arguments left-to-right, then apply builtins when their arguments are values.
So far, we have described three rules for reducing arithmetic expressions in a sequence of steps:
- Any number requires no additional work and is a value
- A builtin may be applied when its arguments have been reduced to values
- When we reach a builtin application, we should reduce its arguments from left-to-right
A sequence of reductions (i.e., steps) that follow these rules is called a **reduction sequence**
Write a reduction sequence for…
(+ (* 3 1) (/ 2 2))
Exercise

Write a reduction sequence for…

\((+ (* 3 1) (/ 2 2))\)

\(\rightarrow (+ (* 3 1) (/ 2 2))\)

\(\rightarrow (+ 3 (/ 2 2))\)

\(\rightarrow (+ 3 1)\)

\(\rightarrow 4 ;; this is a value, computation stops.\)
So far, we have only handled arithmetic. Let’s also add if and booleans to our language. It may also be useful to add builtin comparison operators.
**IfArith**, is a language consisting of numbers, booleans, and arithmetic expressions (plus equality testing), along with if

Number ::= 0 | 1 | ...
Bool ::= #t | #f
Value ::= Number | Bool
Expr ::= Value
| (+ expr expr)
| (* expr expr)
| (/ expr expr)
| (= expr expr)
| (if expr expr expr)

We have already covered the highlighted subset
This grammar is in **EBNF (Extended Backus-Naur form)**

Number ::= 0 | 1 | ...
Bool ::= #t | #f
Value ::= Number | Bool
Expr ::= Value
    | (+ expr expr)
    | (* expr expr)
    | (/ expr expr)
    | (= expr expr)
    | (if expr expr expr)
Textual reduction for = happens similarly to + and etc…, except it produces a boolean rather than a number.

\[
(= 1 (+ 2 3))
\rightarrow (= 1 5)
\rightarrow #f
\]

\[
(if \ (= \ (* \ 1 \ (+ \ 2 \ 3)) \ 5) \ 0 \ 1)
\rightarrow (if \ (= \ (* \ 1 \ 5) \ 5) \ 0 \ 1)
\rightarrow (if \ (= \ 5 \ 5) \ 0 \ 1)
\rightarrow (if \ #f \ 0 \ 1) \;; \ what \ next?
\]
Q: What happens when you mess up the types?
A: This is one way in which this lecture is inspecific—we have several choices.

For now, we will say that terms that are “ill typed” get stuck, i.e., have no successor states. Later on, we will build type theory to show that well-typed terms do not get stuck.

\[
\begin{align*}
(+ (* 1 2) (= 3 4)) \\
\rightarrow (+ 2 (= 3 4)) \\
\rightarrow (+ 2 \text{ #f}) \leftarrow \text{☠☠☠}; \text{ can’t make any progress}
\end{align*}
\]
Last, to evaluate an if: first evaluate its guard, then evaluate either the true or false branch based on the guard’s value.

```
(if (= 1 (+ 0 1)) (* 2 3) (* 3 1))
-> (if (= 1 1) (* 2 3) (* 3 1))
-> (if #t (* 2 3) (* 3 1))
-> (* 2 3) ;; replace with true branch
-> 6
```

```
(if (= 1 (+ 1 1)) (* 2 3) (* 3 1))
-> (if (= 1 2) (* 2 3) (* 3 1))
-> (if #f (* 2 3) (* 3 1))
-> (* 3 1) ;; false
-> 6
```
(Informal) Textual Reduction for IfArith:
- Any number/bool requires no additional work and is a value
- A builtin (including =) may be applied when its arguments have been reduced to values and are of the right type
- When we reach a builtin application, we should reduce its arguments from left-to-right
- To reduce if, first reduce the guard, then reduce the appropriate branch
A note on state...

In the textual reduction style, we transform a whole program to another whole program. Thus, the state of the computation is kept in the current string representing the program.
Looking Forward...

This lecture was an introduction to term-rewriting-style formalisms we will learn later on. IfArith is a tiny sub-Turing-complete language we will see again. With the addition of just a single construct, lambdas (i.e., functions), we will achieve a Turing-complete language!

The textual reduction style can capture arbitrarily-expressive language features! But it is way too slow for a real implementation, so we use it as ground truth that is simple to understand. Then we refine to make it fast!
Case Splitting and Lists Intro

CIS352 — Spring 2021
Kris Micinski
Cond

- Cond allows multiple guards to be checked
- \( \text{(cond } \text{guard}_0 \text{ body}_0) \)
  \( \text{guard}_1 \text{ body}_1 \)
  \( \ldots \)
  \( \text{else body}_{\text{else}} \) \(;; \) optional
- Checks each guard sequentially, evaluates first body

\[
\text{(define (foo x)}
  \text{(cond } [(= x 42) 1]
  [>(x 0) 2]
  [\text{else } 3])\)
\]
Exercise

The absolute value of a number x is:
- x is x is greater than 0
- 0 if x = 0
- -x if x is less than 0

Translate this definition into a function using cond
Exercise

The absolute value of a number x is:

- x is greater than 0
- 0 if x = 0
- -x if x is less than 0

Translate this definition into a function using cond

```
(define (abs x)
  (cond [(> x 0) x]
        [(= x 0) 0]
        [(< x 0) (- x)]))
```
Exercise

Say we have the following:

\[
\text{(cond } [g_0 \ b_0] \\
[g_1 \ b_1] \\
\ldots \\
[\text{else } b_{\text{else}}])
\]

How can we rewrite the above to use only if?
Exercise

Say we have the following:

\[
\text{(cond [\(g_0\) \(b_0\)]
\[g_1\) \(b_1\]
\[
\ldots
\]
\[\text{else}\) \(b_{\text{else}}\])}
\]

How can we rewrite the above to use only if?

\[
\text{(if \(g_0\) \(b_0\)
\(\text{(if \(g_1\) \(b_1\)
\[
\ldots
\]
\(\text{(if \(g_{n-1}\) \(b_{n-1}\) \(b_{\text{else}}\)) \ldots\))}
\]
The function **cons** builds a cons cell / pair

\[
(\text{cons } 0 \ 1)
\]

![Diagram of a cons cell with 0 and 1]
The function \texttt{car} gets the left element \((\texttt{car} \ (\texttt{cons} \ 0 \ 1))\) is 0
The function **cdr** gets the right element

\[(\text{cdr } (\text{cons } 0 1))\] is 1
(cdr (cons 0 1)) is 1

The names **car** and **cdr** come from the original implementation of LISP on the IBM 704.
Lists

• Racket has lists—sequences of cons cells ending w/ ‘()  
• The empty list (or “null”) is special, ‘()  
• Many ways to build them  
  • (list 1 2 3) ;; Variadic function  
  • ‘(1 2 3) ;; Datum representation  
• There are three operations on lists  
  • empty?/null?  
  • first/car  
  • rest/cdr
Lists continued…

- Using empty?, car, and cdr, we can write many utilities
  - All definable ourselves, also in Racket by default
  - (length l) — Length of l
  - (list-ref l i) — Get ith element of list (0-indexed)
  - (append l0 l1) — Append l1 to the end of l0
  - (reverse l) — Reverse the list
  - (member l x) — Check if x is in l
Exercise

Using cond, write a function that takes a list l and an index x and returns...

• The first element if $x = 0$
• The second element if $x = 1$
• The third element if $x = 2$
• Otherwise return ‘unknown’
Case Splitting and Lists Intro

CIS352 — Spring 2021
Kris Micinski