Evaluating Expressions

Expressions can *in principle* be evaluated by *rewriting* them, i.e., by **replacing** function applications with their values. Rewriting changes an expression's *form* but *not* its *value*.

The evaluation process can be implemented in various ways, e.g.,

- string rewriting
- tree rewriting
- stack evaluation (for postfix expressions)

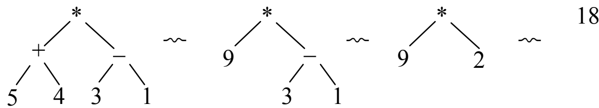
Example:

(5+4) * (3–1)

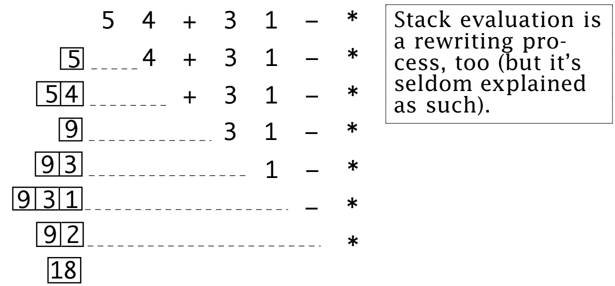
- string rewriting (using '⇝' to indicate steps):

(5+4) * (3–1)	(5+4) * (3–1)
⇝ 9 * (3–1)	⇝ (5+4) * 2
⇝ 9 * 2	⇝ 9 * 2
⇝ 18	⇝ 18

- (Abstract syntax) tree rewriting:



- A postfix expression can be evaluated using a stack:



Each operator has its (language-dependent) rewrite rules.

Example:

True && True	⇝ True
True && False	⇝ False
False && True	⇝ False
False && False	⇝ False

Example:

0 + 0	⇝ 0
0 + 1	⇝ 1
0 + 2	⇝ 2

…

(Of course, specification ≠ implementation!)

Evaluation policies

Expressions are evaluated by repeated application of rewrite rules.

Each application of a rewrite rule is called a ***reduction***.

An expression to which a rewrite rule can be applied is a **red**ucible **ex**pression, or ***redex***.

An expression containing no redexes is said to be in ***normal form***.

Example: the redexes of

3*5 + 8/2

are

3*5 and 8/2

The expression

3*5 + 8/2

is not itself a redex, because addition (+) requires both of its arguments to be in ***normal form***.

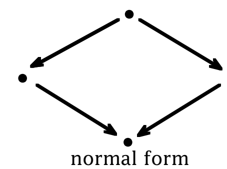
When an expression contains two or more redexes, the choice of the redex(es) to be reduced is governed by an *evaluation policy*.

A few examples of evaluation policies:

- *leftmost*: reduce the leftmost redex
- *rightmost*: reduce the rightmost redex
- *parallel*: reduce all (or several) redexes simultaneously
- *innermost*: reduce a redex that contains no other redex
- *outermost*: reduce a redex that is not contained in any other redex

The **Church-Rosser Theorem**:

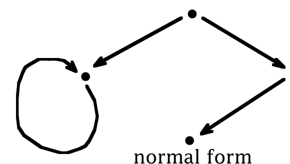
For any given expression, ***all*** reduction sequences that reduce the expression to normal form yield the ***same normal form***.



The choice of evaluation policy may affect *whether* a result is obtained, but not *what* the result's value is.

The Church-Rosser theorem guarantees that expressions can be evaluated in parallel with no risk of altering the result.

It says nothing at all about expressions and reduction sequences that fail to terminate:



Moreover, the theorem holds *only* for *pure* expressions, that is, for expressions that are *referentially transparent*.

Referential Transparency

An expression *E* is referentially transparent iff

- the only thing that matters about *E* is its value
- replacing any subexpression of *E* by any other expression of equal value has *no effect* on the value of *E*.
- every occurrence of *E* in the scope of its variables has the same value.

This same property is at the root of expressions' modularity (slide 10), which makes them easy to understand and to reason about.

In mathematics, expressions are always referentially *transparent*.

In many programming languages, however, "expressions" can be referentially *opaque*.

For example, C allows

```
int i = 3; int j = i++ - i++;
```

Assuming left-to-right evaluation,

- the first occurrence of `i++` has value 3, and the second has value 4; the value of `i++ - i++` is −1.
- replacing the first occurrence of `i++` with its value gives `3 - i++`, whose value is 0.

The "villain" here is that `(++)` not only returns a value but also ***updates*** its argument.

Functional Programming

Programming languages in which expression evaluation —i.e., the application of functions to arguments— plays an especially central rôle are known as *functional* languages.

The two functional languages used in the Sethi textbook —ML and Lisp— are mostly functional, but their inclusion of update operations destroys their referential transparency. Hence they are not purely functional.

In this course, we use neither ML nor Lisp. Instead, we use the *pure* functional language Haskell, in which referential transparency is absolutely preserved.

Haskell's purity clarifies the contrast between the functional paradigm and the imperative paradigm.

Strict adherence to the principle of referential transparency has both benefits and costs. It poses interesting challenges to both

- programmers formulating problem solutions
- implementors striving for efficiency.

Hugs98

For executing Haskell programs, we'll use a programming environment called HUGS98.

> On CS Dept. workstations:
>
> /usr/bin/hugs
>
> To download Hugs for other platforms, see
>
> http://haskell.org/hugs/.

HUGS98 is a command-line interpreter which accepts the user's commands, and displays its responses to those commands, in its *worksheet*.

Among the commands HUGS98 interprets are Haskell expressions, which it evaluates (keyboard inputs are shown here in *italic*):

...“splash screen” omitted...

```
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
Type :? for help
Prelude> 42
42
Prelude> 21*2
42
Prelude>
```

So 42 and 21*2 are Haskell expressions.

`Prelude.hs`, a file of standard definitions, is loaded automatically when HUGS is launched. It defines both ordinary named functions and operators.

For operators, `Prelude.hs` defines not only their meanings but also their precedences and associativities:

```
infixr 9  .
infixl 9  !!
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`, :%, %
infixl 6  +, -
infixr 5  :, ++
infix  4  ==, /=, <, <=, >=, >, `elem`, `notElem`
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

Operator (/) is for floating-point division; (`div`) is for integer division.

```
Prelude> 9 `div` 2     (note the back-quotes!)
4
Prelude> 9 / 2
4.5
```

Operator (++) is for concatenating strings:

```
Prelude> "Hello" ++ ", " ++ "world!"
"Hello, world!"
```

The relational operators (==), (/=), (<), (<=), (>=), (>) are non-associative:

```
Prelude> 1 < 2 == 3 <= 4
ERROR: Ambiguous use of "(==)" with "(<=)"
Prelude> (1 < 2) == (3 <= 4)
True
```