Expression trees and S-expressions Representing the structure of programming languages

Theory of Programming Languages
Computer Science Department
Wellesley College

Table of contents

Expression trees

- The most common kind of trees that we will manipulate in this course are trees that represent the structure of programming language expressions (and other kinds of program phrases).
- In this lecture we begin to explore some of the concepts and techniques used for describing and representing expressions.

EL: A simple expression language

Integer Expressions

An EL integer expression is one of:

- an *intlit* — an integer literal (numeral) *num*;
- a *variable reference* — a reference to an integer variable named *name*
- an *arithmetic operation* — an application of a *rator*, in this case a binary *arithmetic operator*, to two integer *rand* expressions, where an arithmetic operator is one of:
  - addition,
  - subtraction,
  - multiplication,
  - division,
  - remainder;
- a *conditional* — a choice between integer *then* and *else* expressions determined by a boolean *test* expression.

EL Boolean expressions

An EL boolean expression is one of:

- a *boollit* — a boolean literal *bool* (i.e., a true or false constant);
- a *negation* — the negation of a boolean expression *negand*;
- a *relational operation* — an application of *rator*, in this case a binary *relational operator*, to two integer *rand* expressions, where a relational operator is one of:
  - less-than,
  - equal-to,
  - greater-than;
- a *logical operation* — an application of a *rator*, in this case a binary *logical operator*, to two boolean *rand* expressions, where a logical operator is one of:
  - and,
  - or.

The anatomy of an expression

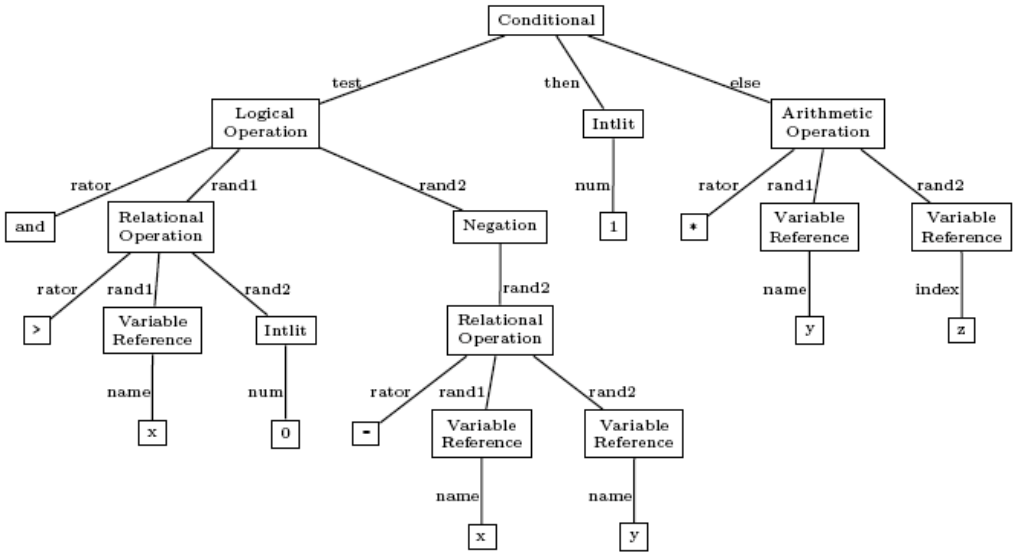
- An integer expression in EL can be constructed out of various kinds of components. Some of the components, like integer literals, variable references, and arithmetic operators, are primitive — they cannot be broken down into subparts.
- Other components, such as arithmetic operations and conditional expressions, are compound — they are constructed out of constituent components.
- The components have names; e.g., the subparts of an arithmetic operation are the rator (short for "operator") and two rands (short for "operands"), while the subexpressions of the conditional expression are the test expression, the then expression, and the else expression.

Abstract grammars: A wiring chart for expressions

- The structural description given above constrains the ways in which integer and boolean expressions may be "wired together."
  - Boolean expressions can appear only as the test expression of a conditional, the negand of a negation, or the operands of a logical operation.
  - Integer expressions can appear only as the operands of arithmetic or relation operations, or as the then or else expressions of a conditional.
- A specification of the allowed wiring patterns for the syntactic entities of a language is called a grammar.
- The above description is said to be an abstract grammar because it specifies the logical structure of the syntax but does not give any indication how individual expressions in the language are actually written down in a concrete form.

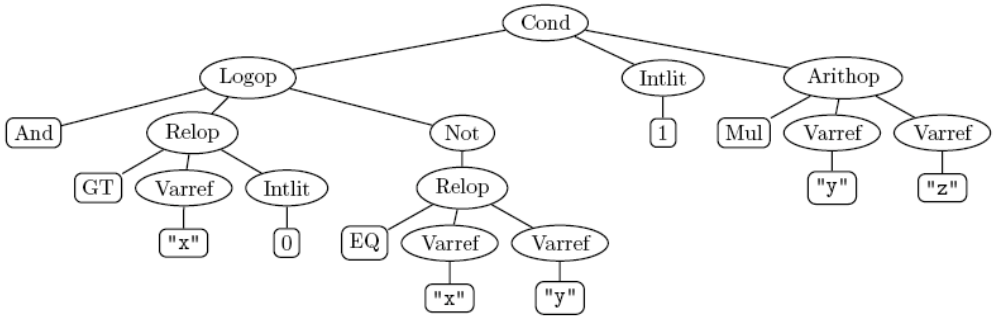
Abstract syntax trees

Parsing an expression with an abstract grammar results in a value called an abstract syntax tree (AST).



Recasting an AST as a sum-of-product tree

We can easily recast any AST as a sum-of-product tree by dropping the edge labels and fixing the left-to-right order of components for compound nodes.



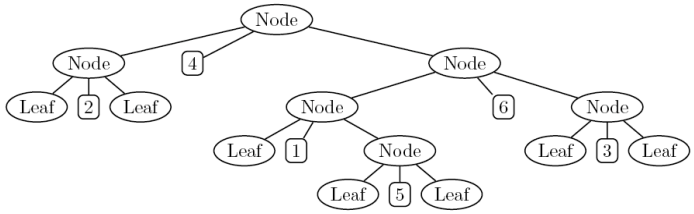
OCaml data type declarations for our AST

Based on this observation, we can describe any EL expressions using the following OCaml data type declarations:

```
type intExp = (* integer expressions *)
    Intlit of int (* value *)
  | Varref of string (* name *)
  | Arithop of arithRator * intExp * intExp (* rator, rand1, rand2 *)
  | Cond of boolExp * intExp * intExp (* test, then, else *)

and boolExp = (* boolean expressions *)
    Boollit of bool (* value *)
  | Not of boolExp (* negand *)
  | Relop of relRator * intExp * intExp (* rator, rand1, rand2 *)
  | Logop of logRator * boolExp * boolExp (* rator, rand1, rand2 *)

and arithRator = Add | Sub | Mul | Div | Rem (* arithmetic operators *)

and relRator = LT | EQ | GT (* relational operators *)

and logRator = And | Or (* logical operators *)
```

The parsing problem

Consider the binary tree:



We can create this in OCaml using constructors:

```
Node(Node(Leaf, 2, Leaf),
     4,
     Node(Node(Leaf, 1, Node(Leaf, 5, Leaf)),
          6,
          Node(Leaf, 3, Leaf))
```

But we'd prefer to use more concise tree notations like:

```
((* 2 *) 4 ((* 1 (* 5 *)) 6 (* 3 *))) ; ''compact''
((2) 4 ((1 (5)) 6 (3)))  ; ''dense''
```


Another example

As another example, consider the sample EL integer expression tree from the previous page. Rather than express it via OCaml constructors, we'd like to use a more concise expression notation. Here are some examples:

```
if x>0 && !(x=y) then 1 else y*z ; Standard infix notation

(if ((x > 0) && (! (x = y))) then 0 else (y * z)) ; Fully
   parenthesized infix notation

x 0 > x y = ! && (1) (y z *) if; Postfix notation

if && > x 0 ! = x y 1 * y z ; Prefix notation

(if (&& (> x  0) (! (= x y))) 1 (* y z)) ; Fully
   parenthesized prefix notation
```

The parsing problem

- To use *any* character-based notation for binary trees and EL expressions it is necessary to decompose a character string using one of these notations into fundamental tokens and then parse these tokens into the desired OCaml constructor tree.
- The problem of transforming a linear character string into a constructor tree is called the parsing problem.

Overview of S-expressions

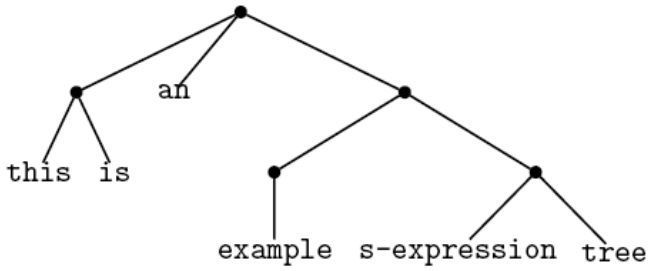
- A symbolic expression (s-expression for short) is a simple notation for representing tree structures using linear text strings containing matched pairs of parentheses.
- Each leaf of a tree is an atom, which (to first approximation) is any sequence of characters that does not contain a left parenthesis (‘`(`’), a right parenthesis (‘`)`’), or a whitespace character (space, tab, newline, etc.).
- Examples of atoms include `x`, `this-is-an-atom`, `anotherKindOfAtom`, `17`, `3.14159`, `4/3*pi*r^2`, `a.b[2]%3`, `'Q'`, and `"a (string) atom"`.

Nodes of an s-expression tree

A node in an s-expression tree is represented by a pair of parentheses surrounding zero or s-expressions that represent the node's subtrees. For example, the s-expression

```
((this is) an ((example) (s-expression tree)))
```

designates the structure depicted below:



Enhancing the readability of an s-expression tree

Whitespace is necessary for separating atoms that appear next to each other, but can be used liberally to enhance (or obscure!) the readability of the structure. Thus, the above s-expression could also be written as

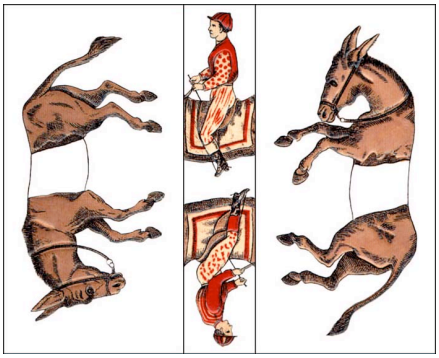
```
((this is)
 an
 ((example)
  (s-expression
   tree)))
```

or (less readably) as

```
                (
     (           this
is) an  (    (          example
           ) (
s-expression tree        )
  )
           )
```

A simple solution to the parsing problem

- We shall see that s-expressions are an exceptionally simple and elegant way of solving the parsing problem — translating string-based representations of data structures and programs into the tree structures they denote.
- For this reason, all the mini-languages we study later in this course have a concrete syntax based on s-expressions.



Representing s-expressions in OCaml

As with any other kind of tree-shaped data, s-expressions can be represented in OCaml as values of an appropriate data type.

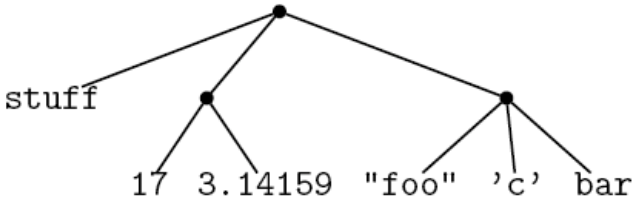
```
type sexp =
    Int of int
  | Flt of float
  | Str of string
  | Chr of char
  | Sym of string
  | Seq of sexp list
```

S-expression nodes

The nodes of s-expression trees are represented via the `Seq` constructor, whose `sexp list` argument denotes any number of s-expression subtrees. For example,

```
(stuff (17 3.14159) ("foo" 'c' bar))
```

which would be expressed in general tree notation as

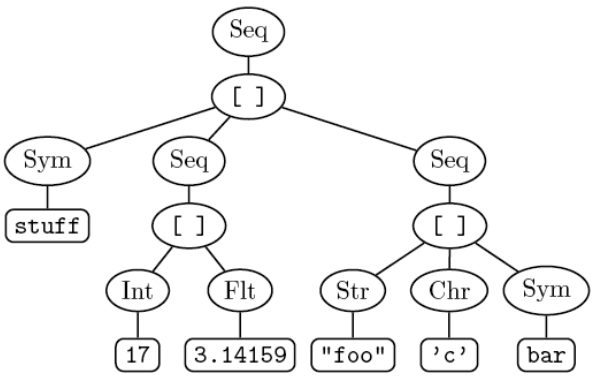


OCaml *s-expression equivalent*

This s-expression can be written in in OCaml ascolorblue

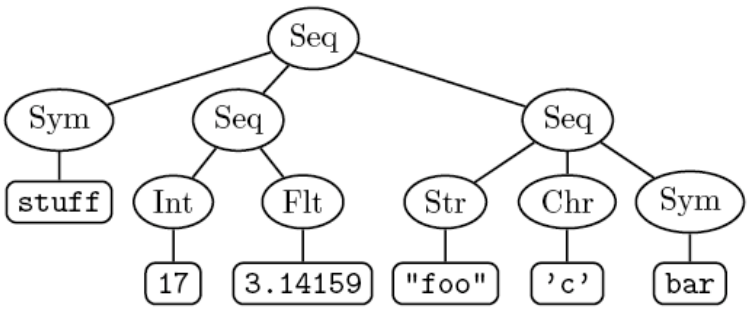
```
Seq [Sym("stuff");
     Seq [Int(17); Flt(3.14159)];
     Seq [Str("foo"); Chr('c'); Sym("bar")]]
```

which corresponds to the following constructor tree:



A somewhat simplier representation

In the constructor tree, nodes labeled `[ ]` represent lists whose elements are shown as the children of the node. Since it's cumbersome to write such list nodes explicitly, we will often omit the explicit `[ ]` nodes and instead show `Seq` nodes as having any number of children:



The `Sexp` *module in* `/cs251/util/Sexp.ml`

```
module type SEXP = sig

  (* The sexp type is exposed for the world to see *)
  type sexp =
      Int of int
    | Flt of float
    | Str of string
    | Chr of char
    | Sym of string
    | Seq of sexp list

  exception IllFormedSexp of string
  (* This exception is used for all errors in s-expression manipulation *)

  val stringToSexp : string -> sexp
  (* (stringToSexp <str>) returns the sexp tree represented by the s-expression
     <str>. Raise an IllFormedSexp exception if <str> is not a valid
     s-expression string. *)

  val stringToSexps : string -> sexp list
  (* (stringToSexps <str>) returns the list of sexp trees represented by <str>, which
     is a string containing a sequence of s-expressions. Raise an IllFormedSexp
     exception if <str> not a valid representation of a sequence of s-expressions. *)

  val fileToSexp : string -> sexp
  (* (fileToSexp <filename>) returns the sexp tree represented by the s-expression
     contents of the file named by <filename>. Raises an IllFormedSexp exception
     if the file contents is not a valid s-expression. *)
```

The Sexp *module continued*

```
  val sexpToString : sexp -> string
  (* (sexpToString <sexp>) returns an s-expression string representing <sexp> *)

  val sexpToString' : int -> sexp -> string
  (* (sexpToString' <width> <sexp>) returns an s-expression string representing
     <sexp> in which an attempt is made for each line of the result to be
     <= <width> characters wide. *)

  val sexpsToString : sexp list -> string
  (* (sexpsToString <sexps>) returns string representations of the sexp trees
     in <sexps> separated by two newlines. *)

  val sexpToFile : sexp -> string -> unit
  (* (sexpsToFile <sexp> <filename>) writes a string representation of <sexp>
     to the file name <filename>. *)

  val readSexp : unit -> sexp
  (* Reads lines from standard input until a complete s-expression has been
     found, and returns the sexp tree for this s-expresion. *)

end
```


Invocations of the functions from Sexp

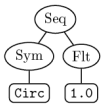
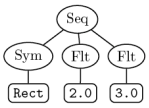
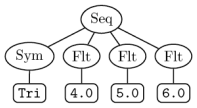
```
# let s = Sexp.stringToSexp "(stuff (17 3.14159) (\"foo\" 'c' bar))";;
val s : Sexp.sexp =
  Sexp.Seq
   [Sexp.Sym "stuff"; Sexp.Seq [Sexp.Int 17; Sexp.Flt 3.14159];
    Sexp.Seq [Sexp.Str "foo"; Sexp.Chr 'c'; Sexp.Sym "bar"]]

# Sexp.sexpToString s;;
- : string = "(stuff (17 3.14159) (\"foo\" 'c' bar))"

# Sexp.sexpToString' 20 s;;
- : string =
"(stuff (17 3.14159)\n       (\"foo\" 'c'\n              bar\n              )\n       )"

# let ss = Sexp.stringToSexps "stuff (17 3.14159) (\"foo\" 'c' bar)";;
val ss : Sexp.sexp list =
  [Sexp.Sym "stuff"; Sexp.Seq [Sexp.Int 17; Sexp.Flt 3.14159];
   Sexp.Seq [Sexp.Str "foo"; Sexp.Chr 'c'; Sexp.Sym "bar"]]

# Sexp.sexpsToString ss;;
- : string = "stuff\n\n(17 3.14159)\n\n(\"foo\" 'c' bar)"

# Sexp.readSexp();;
(a b
   (c d e)
   (f (g h))
   i)
- : Sexp.sexp =
Sexp.Seq
 [Sexp.Sym "a"; Sexp.Sym "b";
  Sexp.Seq [Sexp.Sym "c"; Sexp.Sym "d"; Sexp.Sym "e"];
  Sexp.Seq [Sexp.Sym "f"; Sexp.Seq [Sexp.Sym "g"; Sexp.Sym "h"]];
  Sexp.Sym "i"]
```

The `Sexp` *module continued*

We will mainly use s-expressions for representing the trees implied by sum-of-product data type constructor invocations in a standard format. We will represent a tree node with tag *tag* and subtrees $t_1 \ldots t_n$ by an s-expression of the form:

(tag <s-expression for t_1> ... <s-expression for t_n>)

For instance, consider the representations of `fig` values:

Constructor Invocation	`Circ 1.0`	`Rect (2.0, 3.0)`	`Tri(4.0, 5.0, 6.0)`
Constructor Tree			
S-expression Tree			
S-expression	`(Circ 1.0)`	`(Rect 2.0 3.0)`	`(Tri 4.0 5.0 6.0)`

Using s-expression representations

To use the s-expression representation of figures, we need a way to convert between `fig` constructor trees and s-expression constructor trees:

```
let toSexp fig =
  match fig with
    Circ r -> Seq [Sym "Circ"; Flt r]
  | Rect (w,h) -> Seq [Sym "Rect"; Flt w; Flt h]
  | Tri (s1,s2,s3) -> Seq [Sym "Tri"; Flt s1;
                           Flt s2; Flt s3]
let fromSexp sexp =
  match sexp with
    Seq [Sym "Circ"; Flt r] -> Circ r
  | Seq [Sym "Rect"; Flt w; Flt h] -> Rect (w,h)
  | Seq [Sym "Tri"; Flt s1; Flt s2; Flt s3] ->
                                    Tri (s1,s2,s3)
  | _ -> raise (Failure
                ("Fig.fromSexp -- can't handle sexp:\n"
                       ^ (sexpToString sexp)))
```

Using our new conversion tools

We use `toSexp` and `fromSexp` in any context where we wish to interactively manipulated `fig` values specified in files or keyboard input from users. For example, suppose we want to interactively scale figures as shown below:

```
# Fig.interactiveScale();;
Enter a scaling factor> 2.3

Enter a sequence of figures
  (in s-expression format) on one line>
(Circ 1.0) (Rect 2.0 3.0) (Tri 4.0 5.0 6.0)

Here are the scaled results:
(Circ 2.3)
(Rect 4.6 6.9)
(Tri 9.2 11.5 13.8)
- : unit = ()
```

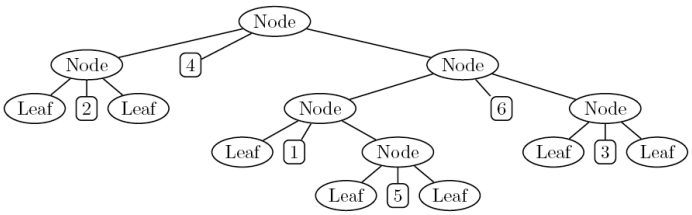

Behind the curtain

Here's how we would defined `interactiveScale` in OCaml:

```
let interactiveScale () =
  let _ = StringUtils.print "Enter a scaling factor> " in
  let s = read_float() in
  let _ = StringUtils.println
            "\nEnter a sequence of figures
             (in s-expression format) on one line> " in
  let line = read_line() in
  let figs = map fromSexp (stringToSexps line) in
  let scaled_figs = map (scale s) figs in
  let _ = StringUtils.println
              "\nHere are the scaled results:" in
     for_each (fun fig ->
        StringUtils.println
          (sexpToString (toSexp fig))) scaled_figs
```

And what about binary trees?

Here is our binary tree example:



Using the above conventions, this would be represented via the s-expression:

```
(Node (Node (Leaf) 2 (Leaf))
      4
      (Node (Node (Leaf) 1 (Node (Leaf) 5 (Leaf)))
            6
            (Node (Leaf) 3 (Leaf))))  ; ‘‘verbose’’
```


A more economical representation

But this is not a very compact representation! As mentioned above, we can often develop more compact representations for particular data types. For instance, here are other s-expressions representing the binary tree above:

```
((* 2 *) 4 ((* 1 (* 5 *)) 6 (* 3 *)))  ; ‘‘compact’’

((2) 4 ((1 (5)) 6 (3)))  ; ‘‘dense’’
```

In the following slide we develop functions that convert between binary trees and these more concise s-expression notations.

Binary trees to verbose s-expressions and back

```
let rec toVerboseSexp eltToSexp tr =

let rec fromVerboseSexp eltFromSexp sexp =

let rec toCompactSexp eltToSexp tr =

let rec fromCompactSexp eltFromSexp sexp =

let rec toDenseSexp eltToSexp s =

let rec fromDenseSexp eltFromSexp sexp =
```