Discrete Mathematics, Lecture 27.

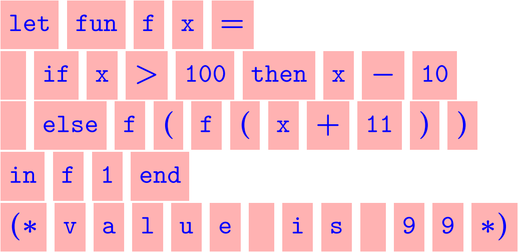
2024-02-07

Abstract Syntax Trees

Concrete syntax: strings of symbols

- possibly including symbols to disambiguate the semantics (brackets, white space, *etc*),
- or that have no semantic content (*e.g.* syntax for comments).

For example, an ML expression:

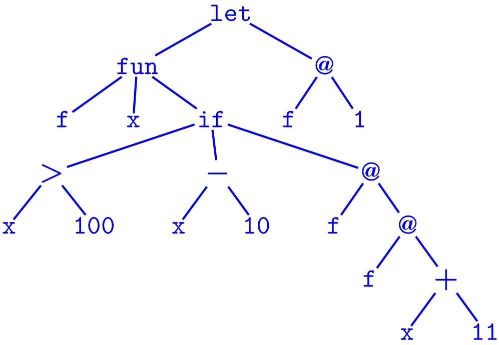


```
let fun f x =
  if x > 100 then x - 10
  else f ( f ( x + 11 ) )
in f 1 end
(* value is 99 *)
```

Abstract syntax: finite rooted trees

- vertexes with n children are labelled by operators expecting n arguments (n-**ary** operators) – in particular leaves are labelled with $\mathbf{0}$-ary (nullary) operators (constants, variables, *etc*)
- label of the root gives the ‘outermost form’ of the whole phrase

E.g. for the ML expression on Slide 25:



Regular expressions (concrete syntax)

over a given alphabet Σ.

Let Σ' be the 6-element set $\{\epsilon, \emptyset, |, {}^*, (,)\}$ (assumed disjoint from Σ)

$$U = (\Sigma \cup \Sigma')^*$$

axioms: $\dfrac{}{a}\,(a \in \Sigma) \qquad \dfrac{}{\epsilon} \qquad \dfrac{}{\emptyset}$

rules: $\dfrac{r}{(r)} \qquad \dfrac{r \quad s}{r|s} \qquad \dfrac{r \quad s}{rs} \qquad \dfrac{r}{r^*}$

(where $a \in \Sigma$ and $r, s \in U$)

Some derivations of regular expressions (assuming $a, b \in \Sigma$)

$$\dfrac{\overline{\epsilon} \qquad \dfrac{\overline{a} \qquad \dfrac{\overline{b}}{b^*}}{ab^*}}{\epsilon|ab^*}$$

$$\dfrac{\dfrac{\overline{\epsilon} \qquad \overline{a}}{\epsilon|a} \qquad \dfrac{\overline{b}}{b^*}}{\epsilon|ab^*}$$

$$\dfrac{\overline{\epsilon} \qquad \dfrac{\dfrac{\overline{a} \qquad \overline{b}}{ab}}{ab^*}}{\epsilon|ab^*}$$

$$\dfrac{\overline{\epsilon} \qquad \dfrac{\dfrac{\overline{a} \qquad \dfrac{\dfrac{\overline{b}}{b^*}}{(b^*)}}{a(b^*)}}{(a(b^*))}}{\epsilon|(a(b^*))}$$

$$\dfrac{\dfrac{\dfrac{\overline{\epsilon} \qquad \overline{a}}{\epsilon|a}}{(\epsilon|a)} \qquad \dfrac{\dfrac{\overline{b}}{b^*}}{(b^*)}}{(\epsilon|a)(b^*)}$$

$$\dfrac{\overline{\epsilon} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\overline{a} \qquad \overline{b}}{ab}}{(ab)}}{(ab)^*}}{((ab)^*)}}{\epsilon|((ab)^*)}$$

Regular expressions (abstract syntax)

The ‘signature’ for regular expression abstract syntax trees (over an alphabet $\boldsymbol{\Sigma}$) consists of

- binary operators $\boldsymbol{Union}$ and $\boldsymbol{Concat}$
- unary operator $\boldsymbol{Star}$
- nullary operators (constants) $\boldsymbol{Null}$, $\boldsymbol{Empty}$ and $\boldsymbol{Sym_a}$ (one for each $\boldsymbol{a \in \Sigma}$).

$(ab) \mid c^*$ $\qquad$ $Union(Concat(Sym_a, Sym_b), Star(Sym_c))$

Regular expressions (abstract syntax)

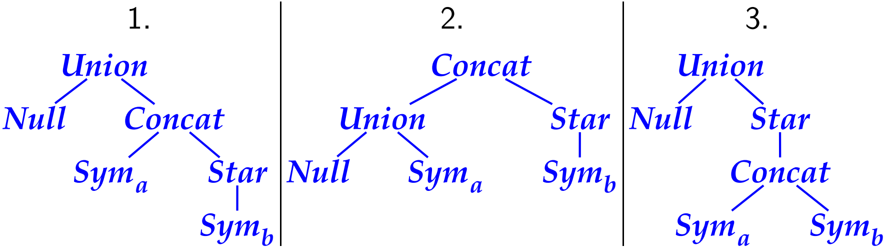
The ‘signature’ for regular expression abstract syntax trees (over an alphabet Σ) as an ML datatype declaration:

```
datatype 'a RE  =  Union of ('a RE) * ('a RE)
                |  Concat of ('a RE) * ('a RE)
                |  Star of 'a RE
                |  Null
                |  Empty
                |  Sym of 'a
```

(the type 'a RE is parameterised by a type variable 'a standing for the alphabet Σ)

Some abstract syntax trees of regular expressions (assuming $a, b \in \Sigma$)

$\varepsilon \mid a(b^*)$



(*cf.* examples on Slide 28)

We will use a textual representation of trees, for example:

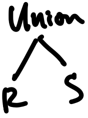
1. $\mathit{Union}(\mathit{Null}, \mathit{Concat}(\mathit{Sym}_a, \mathit{Star}(\mathit{Sym}_b)))$
2. $\mathit{Concat}(\mathit{Union}(\mathit{Null}, \mathit{Sym}_a), \mathit{Star}(\mathit{Sym}_b))$
3. $\mathit{Union}(\mathit{Null}, \mathit{Star}(\mathit{Concat}(\mathit{Sym}_a, \mathit{Sym}_b)))$

Relating concrete and abstract syntax

for regular expressions over an alphabet Σ, via an inductively defined relation $\sim$ between strings and trees:

$$\frac{}{a \sim \mathit{Sym}_a} \qquad \frac{}{\epsilon \sim \mathit{Null}} \qquad \frac{}{\emptyset \sim \mathit{Empty}}$$

$$\frac{r \sim R}{(r) \sim R} \qquad \frac{r \sim R \qquad s \sim S}{r|s \sim \mathit{Union}(R,S)}$$



$$\frac{r \sim R \qquad s \sim S}{rs \sim \mathit{Concat}(R,S)} \qquad \frac{r \sim R}{r^* \sim \mathit{Star}(R)}$$

For example:

$$\begin{aligned}
\epsilon|(a(b^*)) &\sim Union(Null, Concat(Sym_a, Star(Sym_b))) \\
\epsilon|ab^* &\sim Union(Null, Concat(Sym_a, Star(Sym_b))) \\
\epsilon|ab^* &\sim Concat(Union(Null, Sym_a), Star(Sym_b))
\end{aligned}$$

Thus $\sim$ is a ‘many-many’ relation between strings and trees.

- **Parsing:** algorithms for producing abstract syntax trees $parse(r)$ from concrete syntax r, satisfying $r \sim parse(r)$.
- **Pretty printing:** algorithms for producing concrete syntax $pp(R)$ from abstract syntax trees R, satisfying $pp(R) \sim R$.

(See CST IB Compiler construction course.)

We can introduce **operator precedence** and **associativity** conventions for concrete syntax and cut down the pairs in the $\sim$ relation to make it single-valued (that is, $r \sim R \ \& \ r \sim R' \Rightarrow R = R'$). For example, for regular expressions we decree:

> _* binds more tightly than _ _, binds more tightly than _|_

So, for example, the only parse of $\epsilon|ab^*$ is the tree

$$Union(Null, Concat(Sym_a, Star(Sym_b)))$$

Concat (Union (ε , Sym a) , Star (Sym b))

We also decree that the binary operations of concatenation _ _ and union _|_ are left associative, so that for example abc parses as

$$Concat(Concat(Sym_a, Sym_b), Sym_c)$$

(However, the union and concatenation operators for regular expressions will always be given a semantics that is associative, so the left-associativity convention is less important than the operator-precedence convention.)

Note: for the rest of the course we adopt these operator-precedence and associativity conventions for regular expressions and refer to abstract syntax trees using concrete syntax.

Matching

Each regular expression r over an alphabet Σ determines a language $L(r) \subseteq \Sigma^*$. The strings u in $L(r)$ are by definition the ones that **match** r, where

- u matches the regular expression $\overset{(\mathrm{Sym}_a)}{a}$ (where $a \in \Sigma$) iff $u = a$
- u matches the regular expression ϵ iff u is the null string ε
- no string matches the regular expression $\emptyset$
- u matches $\overset{\mathrm{Union}(r,s)}{r|s}$ iff it either matches r, or it matches s
- u matches $\overset{\mathrm{Concat}(r,s)}{rs}$ iff it can be expressed as the concatenation of two strings, $u = vw$, with v matching r and w matching s
- u matches r^* iff either $u = \varepsilon$, or u matches r, or u can be expressed as the concatenation of two or more strings, each of which matches r.

$\text{Concat}(\text{Sym}_a, \text{Union}(\text{Sym}_b, \text{Sym}_c))$

ab

ac

$(ab)^*$

ε, ab, abab, ababab, ...

Inductive definition of matching

$U = \Sigma^* \times \{$regular expressions over $\Sigma\}$

abstract syntax trees

axioms: $\dfrac{}{(a, a)}$ $\qquad \dfrac{}{(\varepsilon, \epsilon)}$ $\qquad \dfrac{}{(\varepsilon, r^*)}$

(a, Sym$_a$) (ε, Null) (ε, Star(r))

rules:

$$\frac{(u, r)}{(u, r|s)} \qquad \frac{(u, s)}{(u, r|s)}$$

$$\frac{(v, r) \qquad (w, s)}{(vw, rs)} \qquad \frac{(u, r) \qquad (v, r^*)}{(uv, r^*)}$$

Concat(r, s)

(No axiom/rule involves the empty regular expression $\emptyset$ – why?)

Examples of matching

Assuming $\Sigma = \{a, b\}$, then:

- $a|b$ is matched by each symbol in Σ
- $b(a|b)^*$ is matched by any string in Σ^* that starts with a 'b'
- $((a|b)(a|b))^*$ is matched by any string of even length in Σ^*
- $(a|b)^*(a|b)^*$ is matched by any string in Σ^*
- $(\epsilon|a)(\epsilon|b)|bb$ is matched by just the strings ε, a, b, ab, and bb
- $\emptyset b|a$ is just matched by a

Some questions

(a) Is there an algorithm which, given a string u and a regular expression r, computes whether or not u matches r?

(b) In formulating the definition of regular expressions, have we missed out some practically useful notions of pattern? $C\,Sym_a = \varepsilon \mid b(a|b)^* \mid a(a|b)(a|b)^*$

(c) Is there an algorithm which, given two regular expressions r and s, computes whether or not they are **equivalent**, in the sense that $L(r)$ and $L(s)$ are equal sets?

(d) Is every language (subset of Σ^*) of the form $L(r)$ for some r?

(a) The answer to question (a) on Slide 38 is ‘yes’: algorithms for deciding such pattern-matching questions make use of finite automata. We will see this next.

(b) If you have used the UNIX utility `grep`, or a text editor with good facilities for regular expression based search, like `emacs`, you will know that the answer to question (b) on Slide 38 is also ‘yes’—the regular expressions defined on Slide 27 leave out some forms of pattern that one sees in such applications. However, the answer to the question is also ‘no’, in the sense that (for a fixed alphabet) these extra forms of regular expression are definable, up to equivalence, from the basic forms given on Slide 27. For example, if the symbols of the alphabet are ordered in some standard way, it is common to provide a form of pattern for naming ranges of symbols—for example `[a-z]` might denote a pattern matching any lower-case letter. It is not hard to see how to define a regular expression (albeit a rather long one) which achieves the same effect. However, some other commonly occurring kinds of pattern are much harder to describe using the rather minimalist syntax of Slide 27. The principal example is **complementation**, $\sim(r)$:

$$u \text{ matches } \sim(r) \quad \text{iff} \quad u \textit{ does not } \text{match } r.$$

It will be a corollary of the work we do on finite automata (and a good measure of its power) that every pattern making use of the complementation operation $\sim(-)$ can be replaced by an equivalent regular expression just making use of the operations on Slide 27. But why do we stick to the minimalist syntax of regular expressions on that slide? The answer is that it reduces the amount of work we will have to do to show that, in principle, matching strings against patterns can be decided via the use of finite automata.