Syntax and grammars, more datatypes, pattern-matching

Reading programs

Fascinating algorithms! Take CS 235, CS 301.

Syntax: Backus-Naur Form (BNF) notation for grammars

Derivations

Derivation Tree
Ambiguity:

>1 derivation of expression
<expr> ::= <num>
| <expr> + <expr>
| <expr> * <expr>

<num> ::= 0 | 1 | 2 | ...

<expr> + <expr> * <expr>

<expr> -> <expr> + <expr>
-> <expr> + <expr> * 3
-> <num> + <expr> * 3
-> 1 + <expr> * 3
-> 1 + 2 * 3

<expr> * <expr>

<expr> -> <expr> * <num>
-> <expr> * 3
-> <expr> + <expr> * 3
-> <num> + <expr> * 3
-> 1 + <expr> * 3
-> 1 + 2 * 3

Dealing with Ambiguity

Prohibit it.
Force parentheses or equivalent.

Racket, S-expressions:
(there is (always an unambiguous) parse tree)

Allow it with:
Precedence by kind of expression (think order of operations)
1 + 2 * 3 means 1 + (2 * 3)

Directional associativity (left, right)
left-associative function application: f 2 3 means ((f 2) 3)

Abstract Syntax Trees (ASTs) in ML
(or expression trees)

A tiny calculator language:

datatype exp = Constant of int
| Negate of exp
| Add of exp * exp
| Multiply of exp * exp

An expression in ML of type exp:
Add (Constant (10+9), Negate (Constant 4))

How to picture the resulting value:
Add
Constant
Negate
19
| | Constant
| 4

Recursive functions for recursive datatypes

Find maximum constant appearing in an expression.
(Program analysis for our tiny language!)

fun max_constant (e : exp) =
Evaluating expressions in the language

We have defined a tiny calculator programming language. Let’s write an interpreter for it!

```fun eval (e : exp) =```

Datatype bindings, so far

Syntax: `datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn`

Type-checking:

- Adds type `t` and constructors `Ci` of type `ti` to static environment
- `Ci v` is a value, i.e., the result “includes the tag”

Evaluation: nothing!

Omit “of `t`” for constructors that are just tags, no underlying data
- Such a `Ci` is a value of type `t`

Case expressions, so far

Syntax: `case e of p1 => e1 | p2 => e2 | ... | pn => en`

Type-checking:

- Type-check `e`. Must have same type as all of `p1` ... `pn`.
- Pattern `C(x1, ..., xn)` has type `t` if datatype `t` includes a constructor: `C of t1 * ... * tn`
- Type-check each `ei` in current static environment extended with types for any variables bound by `pi`.
- Pattern `C(x1, ..., xn)` gives variables `x1`, ..., `xn` types `t1`, ..., `tn` if datatype `t` includes a constructor: `C of t1 * ... * tn`
- All `ei` must have the same type `u`, which is the type of the entire case expression.

Evaluation:

- Evaluate `e` to a value `v`
- If `pi` is first pattern to match `v`, then result is evaluation of `ei` in dynamic environment “extended by the match.”
- Pattern `Ci(x1, ..., xn)` matches value `Ci(v1, ..., vn)` and extends the environment by binding `x1` to `v1` ... `xn` to `vn`
- For “no data” constructors, pattern `Ci` matches value `Ci`
- Pattern `x` matches and binds to any value of any type.
- Exception if no pattern matches.
Hold on to your hats!

Deep truths about ML and patterns.

• Every val-binding and function-binding uses pattern-matching.

• Every function in ML takes exactly one argument

First: extend our definition of pattern-matching...

Pattern-match any compound type

Pattern matching also works for records and tuples:

• Pattern \((x_1, ..., x_n)\) matches any tuple value \((v_1, ..., v_n)\)

• Pattern \(\{ f_1=x_1, ..., f_n=x_n \}\) matches any record value \(\{ f_1=v_1, ..., f_n=v_n \}\)
  (and fields can be reordered)

val-binding patterns

• Remember tuple bindings?

• A val-binding can use a pattern, not just a variable

• Variables are just one kind of pattern.

• Style:
  • Get all/some pieces out of a product/each-of type
  • Usually poor style to put constructor pattern in val binding
    • Raises exception if different variant is there.

Function-argument patterns

A function argument is a pattern

• Match against the argument in a function call

Examples (fantastic style!):

\[
\text{fun } \text{sum_triple} (x, y, z) = x + y + z
\]

\[
\text{fun } \text{full_name} \{ \text{first}=x, \text{middle}=y, \text{last}=z \} =
\]

\[
x ^ ^ = ^ ^ y ^ ^ = ^ ^ z
\]
Convergence!

Takes one int*int*int triple, returns int that is their sum:

```plaintext
fun sum_triple (x, y, z) = x + y + z
```

Takes three int values, returns int that is their sum:

```plaintext
fun sum_triple (x, y, z) = x + y + z
```

Every ML function takes exactly one argument

• "Multi-argument" functions:
  • match a tuple pattern against their single argument
  • Elegant, flexible language design

• Cute and useful things

```plaintext
fun rotate_left (x, y, z) = (y, z, x)
fun rotate_right t = rotate_left(rotate_left t)
```

• “Zero arguments” is the unit pattern ()
  matching the unit value ()

Even more pattern-matching

• Patterns are so much fun!

```plaintext
fun eval e = case e of
  Constant i => i
  Negate e2 => ~ (eval e2)
  Add (e1, e2) => (eval e1) + (eval e2)
  Multiply (e1, e2) => (eval e1) * (eval e2)
```

```plaintext
fun eval (Constant i) = i
| eval (Negate e2) = ~ (eval e2)
| eval (Add (e1, e2)) = (eval e1) + (eval e2)
| eval (Multiply (e1, e2)) = (eval e1) * (eval e2)
```

• Patterns are deep!

• Patterns are recursively structured
  • Just like expressions
  • Nest as deeply as desired
  • Avoid hard-to-read, wordy, nested case expressions

• Full meaning of pattern-matching:
  • compare pattern against value for “same shape”
  • bind variables to the “right parts”

• More precise recursive definition after examples
Examples

- Pattern `a::b::c::d` matches all lists with ________ elements
- Pattern `a::b::c::[]` matches all lists with ________ elements
- Pattern `[a,b,c]` matches all lists with ________ elements
- Pattern `{(a,b), (c,d)}::e` matches all ________

List checkers

```haskell
fun nondec (x::xs) = 
  case xs of 
    ([]) => x <= y andalso nondec xs
  | nondec [] = true
| nondec _ = nondec []

fun nondec [] = true
| nondec [x] = true
| nondec (x::xs) = 
  let val (y::_) = xs
  in 
    x <= y andalso nondec xs
  end
```

List checkers: even better!

```haskell
fun nondec (x::y::zs) = x <= y andalso nondec (y::zs)
| nondec _ = true

fun allsq (x::y::zs) = x*x = y andalso allsq (y::zs)
| allsq _ = true

fun checkl (f, x::y::zs) = 
  f (x,y) andalso checkl (f, y::zs)
| checkl _ = true
```

More examples to come (see code files)

Style

- Nested patterns: elegant, concise code
  - Avoid nested case expressions if nested patterns are simpler
    - Example: checkl and friends
  - Common idiom: match against a tuple of datatypes to compare all
    - Examples: zip3 and multisign

- Wildcards are good style: use instead of variables when you do not need the data
  - Examples: len and multisign
(Most of) the definition of pattern-matching

The **semantics** for pattern-matching takes a pattern \( p \) and a value \( v \) and decides (1) does it match and (2) if so, what variable bindings are introduced.

Definition is elegantly recursive, with a separate rule for each kind of pattern. Some of the rules:

- If \( p \) is a variable \( x \), the match succeeds and \( x \) is bound to \( v \)
- If \( p \) is \( _ \), the match succeeds and no bindings are introduced
- If \( p \) is \( (p_1, \ldots, p_n) \) and \( v \) is \( (v_1, \ldots, v_n) \), the match succeeds if and only if \( p_1 \) matches \( v_1 \), \ldots, \( p_n \) matches \( v_n \). The bindings are the union of all bindings from the submatches
- If \( p \) is \( C \, p_1 \), the match succeeds if \( v \) is \( C \, v_1 \) (i.e., the same constructor) and \( p_1 \) matches \( v_1 \). The bindings are the bindings from the submatch.
- … (there are several other similar forms of patterns)

intuition...

Do you suppose...?

```
datatype int = ... | 0 | 1 | 2 | ...
```

(Efficiency reasons to implement int specially, but could be a datatype.)

No reason we couldn't...

```
datatype nat = Zero | Succ nat
val one = Succ Zero
fun add (Zero, x) = x
    | add (x, Zero) = x
    | add (Succ x, y) = Succ (add (x, y))
```

Got a match?

Semantically equivalent:

```
fun fib n =
    if n = 0 then 1
    else if n = 1 then 1
    else (fib (n - 2)) + (fib (n - 1))
```

```
fun fib n =
    case n of
        0 => 1
        | 1 => 1
        | x => (fib (x - 2)) + (fib (x - 1))
```

```
fun fib 0 = 1
    | fib 1 = 1
    | fib n = (fib (n - 2)) + (fib (n - 1))
```

The truth about true (and false)

```
datatype bool = true | false
```

```
if e1 then e2 else e3  just sugar
```

```
case e1 of
    true => e2
    | false => e3
```

Are you noticing a pattern here?