Patterns Everywhere

Reading programs

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

Non-terminals

Terminal (lexical tokens)

Start symbol: <expr> designates "root"

Source Program

Lexical Analysis

Break up string input into symbols.

Syntax Analysis (Parsing)

Parse stream of symbols into structured representation of program.

Semantic Analysis

Fascinating algorithms!

Take CS 235, CS 301.

Derivations

<expr> := <num>
| <expr> + <expr>
| <expr> * <expr>

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

<expr> => <num>
| 5

<expr> => <expr> + <expr>
| <num> + <expr>
| 1 + <expr>
| 1 + <expr> * <expr>
| 1 + <num> * <expr>
| 1 + 2 * <expr>
| 1 + 2 * <num>
| 1 + 2 * 3
Representing Abstract Syntax Trees (ASTs)
(or expression trees)

A tiny calculator language:

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

An ML expression of type `exp`:

```
Add (Constant (10+9), Negate (Constant 4))
```

Structure of resulting value:

```
Add
  Constant 19
  Negate
    Constant 4
```

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)`

Recursive functions for recursive datatypes

Find maximum constant appearing in an `exp`.

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

Interpreter for tiny calculator language

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

**Datatype bindings, so far**

Syntax:

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

Type-checking:

Adds type \( t \) and constructors \( C_i \) of type \( t_i \rightarrow t \) to static environment

Evaluation: nothing!

Omit “of \( t_i \)” for constructors that are just tags, no underlying data

– Such a \( C_i \) is a value of type \( t \)

**Case expressions, so far**

Syntax:

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

Type-checking:

– Type-check \( e \). Must have same type as all of \( p_1 \ldots p_n \).
  • Pattern \( C \langle x_1, \ldots, x_n \rangle \) has type \( t \) if datatype \( t \) includes a constructor: \( C \) of \( t_1 \times \ldots \times t_n \)
  – Type-check each \( e_i \) in current static environment extended with types for any variables bound by \( p_i \).  
  • Pattern \( C \langle x_1, \ldots, x_n \rangle \) gives variables \( x_1, \ldots, x_n \) types \( t_1, \ldots, t_n \) if datatype \( t \) includes a constructor: \( C \) of \( t_1 \times \ldots \times t_n \)
  – All \( e_i \) must have the same type \( u \), which is the type of the entire case expression.

Evaluation:

– Evaluate \( e \) to a value \( v \)
  – If \( p_i \) is first pattern to match \( v \), then result is evaluation of \( e_i \) in dynamic environment “extended by the match.”  
    • Pattern \( C_i \langle x_1, \ldots, x_n \rangle \) matches value \( C_i \langle v_1, \ldots, v_n \rangle \) and extends the environment by binding \( x_1 \) to \( v_1 \ldots x_n \) to \( v_n \)
    – For “no data” constructors, pattern \( C_i \) matches value \( C_i \)
  • Pattern \( x \) matches and binds to any value of any type.
    – Exception if no pattern matches.
Patterns everywhere

Deep truths about ML and patterns.

1. Every val/fun binding and anonymous fn definition uses pattern-matching.

2. Every function in ML takes exactly one argument

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

val binding patterns

Syntax: a val binding can use any pattern p, not just a variable

val p = e

Type checking:

p and e must have the same type.

Evaluation:

1. Evaluate e to value v.
2. If p matches v, then introduce the associated bindings
   Else raise an exception.

Style:

– Get all/some pieces out of a product/each-of type
– Often poor style to use constructor pattern in val binding.

Pattern-match any compound type

Pattern matching also works for records and tuples:

– Pattern (x1,...,xn)
  matches any tuple value (v1,...,vn)

– Pattern {f1=x1, ..., fn=xn}
  matches any record value {f1=v1, ..., fn=vn}
  (and fields can be reordered)

Parameter patterns

A function parameter is a pattern.

– Match against the argument in a function call.

Examples:

fun sum_triple (x, y, z) = x + y + z
fun full_name {first=x, middle=y, last=z} =
  x ^ " " ^ y ^ " " ^ z
Convergence!

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

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

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

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

Every ML function takes exactly one argument

"Multi-argument" functions:
– Match a tuple pattern against single argument.
– Elegant, flexible language design

Cute and useful things

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

“Zero-argument” functions:
– Match the unit pattern () against single argument.

Even more pattern-matching

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)

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)

Critical: added parens around each pattern, replaced => with =.
– If you mix them up, you’ll get some weird error messages...

Patterns are deep!

Patterns are recursively structured
– Just like expressions
– Nest as deeply as desired
– Avoid hard-to-read, wordy, nested case expressions
Examples of nested list patterns

Pattern `a::b::c::d` matches any list with _______ elements

Pattern `a::b::c::[]` matches any list with _______ elements

Pattern `[a,b,c]` matches any list with _______ elements

Pattern `((a,b),(c,d))::e` matches any _______

List checkers (suboptimal style)

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

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

List checkers (good style)

```
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: see code files

Style

Nested patterns: elegant, concise
  – 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 multsign

Wildcards instead of variables when data not needed
  – Examples: len and multsign
(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)

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

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

datatype bool = true | false

if e1 then e2 else e3

case e1 of  
  true => e2  
  false => e3

Are you noticing a pattern here?