Matches, Exceptions, and Polymorphic Functions

10-25-2004

Opening Discussion

- Last time we talked about I/O in ML. Can anyone remind me of some of the details of I/O? I/O in ML is somewhat different than in Scheme. How is it different and why?
- Given what we have talked about so far in ML, what do you consider the most interesting feature of the language?

Matches

- Patterns can be used in part of a structure called a match in ML. These appear in function definitions, case expressions, and exception handlers.

```
<pattern 1> => <expression 1> |
<pattern 2> => <expression 2> |
...
<pattern N> => <expression N>
```

Matches as Functions

- In addition to the way we have defined functions with fun, they can also be defined with fn and a match.
- The rec is only required if it is a recursive function.
- Without the val this can be used to define anonymous functions.

```
val rec <name> = fn <P1> => <E1> |
<P2> => <E2> | ... | <PN> => <EN>;
```

Case Expressions

- ML has a case construct that takes the form below. Basically the case does pattern matching. Notice how this is significantly more powerful than the switch statements in C/C++/Java because the match can include constants of any type other than reals and can also break up tuples or lists.

```
case <expression> of <match>
```

- The if-then-else in ML is actually a case that matches the patterns true and false.

Exceptions

- Functions that don't produce valid answers for all inputs can (and should) raise exceptions.
- ML has built in exceptions for things like “5 div 0”, “hd(nil:int list)”, or “chr(500)”.
- We can define our own exceptions as new types with the keyword “exception” followed by type names separated by “and”. Type names typically start with a capital letter.
- When we need code to generate an exception we use “raise”.

Exceptions with Parameters

- Sometimes you want more information than just the exception name. In that case you can attach a parameter to the exception with the keyword “of”
- For example “exception Foo of string;” makes Foo an exception type that has a string type for the parameter.
- When we raise one of these we have to provide the parameter so in this case it might be “raise Foo(“Bad Things.”)”

Handling Exceptions

- If exceptions aren't handled, the execution is terminated. If we want to handle an exception we can use the syntax below.
- If the expression raises an exception, the proper matching exception is found and the value will be the expression that match goes to.

```
<expression> handle <match>
```

Local Exceptions

- Exception types can also be declared locally in let expressions.
- The problem with doing this is that you can't handle those exceptions outside of that let expression because you are outside of the scope in which the exception is defined.

Polymorphic Functions

- We saw last time that ML has types that can represent ANY type. These are polymorphic types. One very polymorphic function is the identity function, which given something returns it.
- There are some detailed limitations on polymorphism we won't discuss, but which your book does. Odds are they won't matter to you.

Operators and Polymorphism

- Certain operations prevent polymorphism. They include arithmetic operations, inequalities, boolean operators, string concatenation, and type conversions.
- Other operators allow polymorphism. These include tuple operators, list operators, and equality operators (=, <>).
- The equality operators limit us to equality types. Basically anything that doesn't include functions or reals is an equality type. Equality types have two primes (''a).

Two Forms of Reverse

- As a simple example, we can write two forms of the reverse function. One that uses if and one that uses patterns.
- Notice that the types of these are different. Why should this be the case?
- To avoid this we could use the predicate null (<list>) in the if, but using patterns is even easier.

Minute Essay

- Write a function called cull that takes a predicate and a list. It should call the predicate on each element of the list and if it is true, put it in the list that is returned. Then show how you would call this function with an anonymous function.
- You should make sure you start working on assignment #6 soon.