Basics of SML

Values

- There are basic values which are the building blocks of the language
- Values cannot be reduced to simpler forms
- Values are grouped into categories called types
- e. g. 17:int, ~2:int, 3.14159:real, “Friedman”:string, true:bool

Expressions

- Simple expressions are formed by combining expressions using operators
- Examples of expressions
    - 2, 1+2, (1+2)*(3+4)
    - “Hello”, “Hello” ^ “ World”
    - 5 div 0
    - “Hello” + 3

Evaluation: Computing by calculating

- To run an ML program, we calculate it down to a value
- Values cannot be reduced
- Every value is an expression. It evaluates to itself in 0 steps
- <exp> ==> <val> is our meta-notation meaning that the expression <exp> calculates to the value <val>
- Example
    - 2 ==> 2,
    - 1+2 ==> 3,
    - (1+2)*(3+4) ==> 21,
    - “Hello” + 3 has no value
    - 5 div 0
- The value of an expression is determined by calculating until no further calculations can be done
- For each operator, the semantics will specify how to calculate. In general we calculate the subexpressions and then apply the operation
- One step of the calculation is denoted |->
- Example:
  (1 + 2) * (3 + 4) |-> 3 * (3+4) |-> 3 * 7 |-> 21
- 5 div 0 raises an exception
- Exceptions propagate up so the error will be the final result:
  (5 div 0) *7 |-> (raise DIV) + 1 |-> raise Div
- Defn: An expression is **valuable** iff there is some value that it evaluates to
- We can also do parallel calculations by the principal of referential transparency

Referential Transparency

- Functional languages have the property of “referential transparency”

- That means you can replace any expression with another expression that has an “equal” value without affecting the value of the expression
- Since evaluation has no side-effects, it does not matter in which order sub-expressions are evaluated
- This property supports parallelism, program optimization and reasoning about programs

Types

- the type of an expression is a prediction about the value it will yield if it does yield a value
- For example 5 div 0 has type int but does not yield a value
- We classify expressions according to the properties of the values they can take on
- A type is a static approximation of the run-time value of the program
- Type checking is done before execution in a strongly typed language
- An expression is well-typed if it has at least one type (polymorphism exists when a value has more than one type e.g. list of different types)
- An expression is ill-typed if it has no types (“hello” + 3)
- A type checker determines whether an expression is well-typed and rejects ill-typed expressions at compile time.
- <exp> : <type> denotes that <exp> has type <type>
- Examples:
    - 2 : int,
    - (1+2)*(3+4) : int,
    - “hello” ^ “ world” : string
- a type in general is specified by a collection of values and a collection of operations
- Examples
    - type int:
        - values 0, ~25, 83, …
        - operations: +, *, -, div intToString
    - type string:
        - values: “hello”, “Nathan”,…
        - operations: ^, size, …
    - Type real
        - values 0.0, 3.1415, …
        - operations: +, -, *, /
        - Some of these (+,-,*) are overloaded
        - disambiguated based on context
        -

Classes of expressions - summary

- nonsense (syntactically incorrect) (1 + 2
- syntactically correct “hello” + 3 ==> type error at compile time
- well-typed
- valuable expressions (compute to a value) 5 div 0 not valuable
- values

For each operator there are rules for type checking as well as evaluation. Some examples are:

- Addition Operator
    - syntax: e1 + e2 where e1 and e2 are expressions

    - Type-checking rule: if e1 and e2 have type int, e1+e2 has type int
    - Evaluation: if e1 evaluates to v1 and e2 evaluates to v2, e1+e2 evaluates to v1+v2

- Relational Operator <
    - Syntax: e1 < e2
    - Type-checking rule: if e1 and e2 have type int, e1<e2 has type bool in same static envirionment
    - Evaluation: if e1 evaluates to v1 and e2 evaluates to v2, e1+<e2 evaluates to true if v1<v2 and false otherwise

- Conditionals
    - Syntax: if e1 then e2 else e3, were e1, e2 and e3 are expressions
    - Type-checking rule: if e1 has type book and e2 and e3 have the same type then expression has same type as e2 and e3
    - Evaluation: in current dynamic environment evaluate e1 to get v1. If it v1 is “true”, evaluate e2 and the value of the expression is v2. If v1 is false, evaluate e3 and the value of the expression is e3