Computer Programming I

Avery Karlin

Contents

Primary Textbook:
Teacher:

1 Introduction to OCaml

1.1 Program Design

1. Programming can be viewed as the division into four steps, understanding the program, formalizing the interface, writing test cases, and implementing behavior
   (a) The first involves finding any relevent concepts and how they relate to each other within the program, while the second determines input and output, and their respective formats
   (b) The third determines the correct answer to standard cases, edge cases, and erroneous cases (in which the user is at fault) for how the program should behave, writing test cases in the program
   (c) The fourth step is the programming, often taking recursive decompositions of the problem, dividing it by concept, the relationships between, and the interface to allow easier debugging

1.2 Basic OCaml Programming

1. OCaml supports integer (int), boolean (bool), and string (string) primitive/atomic types, able to be defined into expressions combining the types and operations on them, working by order of operations
   (a) Basic operations on integers include +, -, *, /, mod, string_of_int (converting integer to string), on strings (surrounded by double-quotes) include $\hat{}$(concatenation), on booleans include not, &&, ——, and general comparisons include =, ¡¿ (inequality), ¡, ¡=, ¿, ¿=
2. Models of computation are ways of thinking about how a program executes, allowing prediction of its behavior, such as object or value oriented programming
   (a) Value-oriented is the idea that running expressions reduces it to a value, rather than an action, such as input/output, as the basis of OCaml
   (b) This is designated by the notation $< exp >=>< val >$, called the evalutation of the expression by calculation as an abstract model of the computer calculation
      i. The internal steps are written out by $\|- >< intermediaryexp >$ by order of operations
3. If-then-else constructs, written "if ¡exp¿ then ¡exp¿ else ¡exp¿", evaluating one of the last two based on the initial expression
   (a) Since the constructs are expressions by themselves, the else is required since the construct expression needs a value if the expression is false
4. OCaml programmings are made of declarations and commands, the former defining constants, functions, and expressions purely value oriented, the latter testing and generating output, non-value oriented
   (a) The let declaration is written as "let ¡id¿ = ¡exp¿", binding the identifier to the value, unable to be modified otherwise in the program
      i. It is often annotated by "let ¡id¿ : ¡type¿ = ¡exp¿" for ease of reading, though this is not required
      ii. Let declarations then automatically substitute each subsequent ¡id¿ with the value of the expression
      iii. Redeclaring a variable is called shadowing, such that the each identifier is substituted by the nearest enclosing declaration, though this is not recommended, producing

ambiguity
        A. Shadowing rather than redefining is apparent within the limited scope of some shadows, such that it reverts back to the original
    (b) Let-in constructs are local declarations, written "let ¡id¿ : ¡type¿ = ¡exp¿ in ¡exp¿" only using the decaration within the in expression
        i. This is able to be nested, such that each applies in the next which eventually all apply within a single expression
    (c) Parameterized expressions/functions, declared by "let ¡id¿ (¡id1¿:¡type1¿) (¡id2¿:¡type2¿) : ¡type¿ = ¡exp¿", called by "¡id¿ ¡val1¿ ¡val2¿"
        i. This runs by substituting the function expression for the function call and the variable id into the function
5. OCaml is strongly typed, such that each type is different with strict rules about combination or conversion, called well-typed if it has at least one type, able to be determined by the operations acting on it or the data
    (a) The compiler automatically typechecks for ill-typed expressions which cannot be evaluated due to type contradictions
        i. Comparison operators return an error with two values of different types
        ii. The compiler will also alert a missing argument or an extra argument to some function
    (b) Functions use explicit type definition to improve clarity, abbreviating in the description the input types and the output type by "¡id¿ : ¡in1Type¿ -¿ ¡in2Type¿ -¿ ... -¿ ¡outType¿"
6. OCaml has the "failwith ¡errorString¿", ending the program and printing the error message if the expression is evaluated
7. Commands in OCaml are used for I/O, not calculating a value, but rather having some external effect, only able to be used in the top-level of a program, precluded by ";;" to distinguish commands
    (a) "print_string ¡String¿" is used to print a string in output, ending the string by '
n' for a newline, able to use an expression to evaluate to a string
        i. Similarly, "print_endline" prints the string with an automatic newline, "print_int" for an integer
    (b) The "run_test" command is stored in the Assert library, called by the "open Assert" command at the start, used by "run_test ¡descriptionString¿ ¡boolVal¿"

2 Lists and Recursion

1. Lists are sequences of at least zero values, either [], the empty list/nil, or "item1::item2::[]" as the list of [item1, item2], able to compound to longer lists, where :: is the constructor operator for the list
    (a) Lists of lists are also able to be created, and expressions can be used as items, evaluating to the value added to the list
    (b) Lists are also able to be made by shorthand, "[item1;item2]"
    (c) List types are denoted by "¡primitiveType¿ list", required to be a homogeneous/single-type list, such that double lists are denoted "¡primitiveType¿ list list" and so forth
    (d) OCaml is able to deconstruct expressions by pattern matching, written "begin match ¡exp¿ with — ¡case¿ -¿ ¡return¿ ... end"

        i. Thus, for a list, the two cases to test for are "[]" and "<var1>::<list1>", checking for an empty list, such that if the latter case is found, the ¡var1¿ and ¡list1¿ identifiers are bound to the cooresponding list parts
    (e) List concatnation is performed by the "@" operator
2. Lists in OCaml are defined self-referentially, as an inductive data type, stating how to build an atomic instance, and construct a larger value out of it
    (a) As a result, both calculations and creation of lists are done by structural recursion, using an empty list base case, calculating for the head assuming the tail has been calculated already
        i. This can be determined by writing out each new recursion to analyze how each level of recursion can be used to get the desired result
    (b) Recursive functions in OCaml must be defined by "let rec" at the start instead of just "let"
    (c) OCaml does not ensure that all recursion is structural (taking a smaller portion of the structure towards the base case), providing the possibility for loops or an infinite loop

3 Tuples and Nested Patterns

1. Tuples store a set number of values (at least 2) of different types, able to use expressions or complex types, denoted generally as "(¡val1¿, ¡val¿2, ...¿) : ¡type1¿ * ¡type2¿ ..."
    (a) Pattern matching can be used similarly, though generally only needing a single case, due to only one form of construction
        i. Let constructions can also be used, written "let $(< var1 >, < var2 >) =< var >$ $in < var1/var2 >$" to set $< var >$ to either $< var1 >$ or $< var2 >$ for a two-item tuple
        ii. Nested patterns of lists and tuples can also be combined for more complex pattern matching, testing the patterns in the order listed in the match statement
        iii. If there is no match, it will return a Match_Failure when run, such that it will give a non-exaustive pattern-matching warning during compilation
        iv. In a pattern match, if a variable is not used, it can be replaced by _ to act as a placeholder/wildcard for an unused value
    (b) Empty tuples are represented by () with type of unit, used as the function paramter for functions without proper inputs, though able to be shorthanded as () in that case instead of (¡var¿: unit)
        i. It is also considered the return type for a command function, such that no output is produced

4 User-defined Datatypes

1. User-defined datatypes are needed to represent application specific types, used to avoid issues with primitives such as invalid values, confusion of types, and invalid default operations
    (a) They are defined by "type $< id >= | < val1 > | < val2 >$ ...", where $< id >$ must be all lowercase, while each $< val >$ must start with an uppercase letter, each representing an atomic, discrete value
        i. These are constructed purely by the ¡val¿

(b) Types are also able to be represented as tuples of multiple datatypes, defined by "type $< id >= | < val1 > of < type1 > | < val2 > of < type2 > ...$"
   i. These are constructed by "$< val > (< type\_values >)$", with more complicated match expressions
   ii. Similarly, type abbreviations can rename existing types, such that "type $< id >=< type >$", acting as a substitute for a general constructor name
(c) Types can be defined recursively as well, with some base case, with the other discrete values using the type itself

5 Binary Trees

1. Trees are used in CS either naturally or due to allowing ordering large amounts of data for efficient searching, defined recursively as either empty or a node
   (a) Root nodes are those at the top of the tree, leaf nodes at the bottom, internal nodes as all others, with each node having 0-2 child notes in a binary tree
   (b) Each node stores some value, and a link to each of its child nodes, but not its parent node, generally "$< leftChild >, < val >, < rightChild >$"
   (c) Trees can be traversed by pre-order (left, right, node), in-order (left, node, right), post-order (left, right, node), able to be used to list items or search
2. Trees can allow for easier searching if there are linearly ordered labels, such that binary search tree invarient are such that the right node being greater than, the left less than, Empty as the default, such that search is logarithmic
   (a) Binary search trees are constructed such that the nodes are inserted or deleted in accordance with the ordering
   (b) Inserting is similarly done to searching, while deleting requires a simple case for 0 children and 1 child, but for 2 children, the left maximum leaf replaces the deleted node and is removed from the bottom

6 Generic Functions and Datatypes

1. Generic functions are those where the type doesn't change the function, given type '¡var¿, where '¡var¿ is the type variable, instantiated to any type by the function, but unable to be reassigned to a different type
   (a) This allows the reusing of functions and algorithms, saving time in both coding and debugging, rather than having large numbers of close to identical functions
   (b) Types can also be made generically, written "type $('< var1 >,'< var2 >) < id >$", such that they can be generalized to being made up of any type components, able to be computed by generic pattern matching

7 First-class Functions

1. Since functions are values in value-oriented programming such as OCaml, functions are able to used as parameters for other functions and returned by functions, called the first-order function characteristic

(a) As a result, functions can also be partially applied, giving only a portion of the inputs, such that it returns a function with that variable filled

(b) Anonymous functions are those without a name, written by the keyword fun, with syntax "fun $(< var1 >:< type1 >)(< var2 >:< type2 >)- >< exp >$", acting as values

i. These are able to be bound by a let expression, "let $< funName >:< type1 > - >< type2 > - >< retType >=< anonFunction >$", equivelent to a standard function form

2. As a result, first order functions are able to be combined with generics to produce higher order functions, which are more generalized functions, taking a function and applying it to some generic object

(a) Common examples of this are used to transform a list by some function or to fold a list by some combine function and a base case into a single object

8 Modularity and Abstraction

1. Abstract types are those which bundle the operations and type name with an interface for the functions by which it can be used, but do not allow direct access to the implementation/proper function codes

(a) Modules are independently compilable collections of code, containing abstract data types and their functions, which when providing common data structures or algorithms, are called libraries

i. These are accessed either by ";; open $< Module >$" or by using dot notation for any identifiers given by the module $(< Module > . < Id >)$

(b) Interfaces are created either in an mli file type or a module type signature, the former written by the type name, followed by each function written by "val $< functName >:< input1 > - >< input2 > - >< output >$", where val is used to denote an expected value

i. Module type signatures are written inside an ml file by "module type $< typeName >= sig < typeInterface >$ end", with the interface written as in an mli file

ii. The implementation can be written as a module of the interface as a nonabstract type, by "module $< typeName >:< AbstractTypeName >= struct < typeImplementation >$ end", called an explicitly named module

A. In this case, the ml file in which they are defined must be opened, after which dot notation must be used for each type, rather than either

(c) For each implementation, OCaml checks if it fully complies with the interface, implementing all expected functions, though it is able to have additional ones

2. Abstract types are useful for not showing the specific usage of implementation, allowing any particular means of implementing, using some implementation type combined with an invarient/rule to preserve the type properties

(a) Each choice of representation and invariant has respective advantages and disadvantages in terms of time optimization

9 Homework Notes

1. No multi command conditionals?

2. In needed for complex functions?
3. Can't create an empty user-defined datatype, accessed by begin match similar to regular types (Node (a, b) -¿, etc)
4. Need either variable or wildcard
5. When can you use multiple lines in a single begin match command?
6. Type is not properly implemented in interface, though a function for an empty can be, since functions are values in OCaml
7. Equals checks if the structure is identical, in addition to the data for any object