My sites / 21S-COMSCI131-1 / Exams / Midterm

Spring 2021 - **Week 8**

Spring 2021 - COM SCI131-1 - EGGERT

Started on	Wednesday, 28 April 2021, 11:30 PM PDT
State	Finished
Completed on	Thursday, 29 April 2021, 1:29 AM PDT
Time taken	1 hour 58 mins
Points	107.00/120.00
Grade	**8.92** out of 10.00 (**89**%)

Question **1**
Complete
2.00 points out of 4.00

List the tokens and tokenization rules for the subset of OCaml that was discussed in lecture and used in Homeworks 1 and 2.

```
anonf ::= ‘fun’, {args}, ‘->’, stmt
stmt ::= patternmatch |
        ‘let’, stmt, {‘and’ stmt}, ‘in’ stmt |
        anonf
patternmatch ::= ‘function’, {‘|’, stmt} |
        ‘match’, t, ‘with’, {‘|’, stmt}
namedfunc ::= "let", {args}, "=", stmt

... Not enough time to write a fully formal-ish ruleset, so here’s a less descriptive list to supplement what’s above:
List constructor ::
List concatenation @
==
=
[]
integer arithmetic: +, - , *
```

Feedback:

Missing tokenization rules e.g. comments and newline

Question **2**
Complete
4.00 points out of 4.00

Give an example of how OCaml uses the greedy rule for tokenization. Your example should not involve identifiers or numbers.

“a”,”b”,”c”
is parsed as a three element tuple
string * string * string = ("a", "b", "c")

Parentheses are needed for tuples within a tuple, e.g.
("a", "b"), "c"
is parsed as
(string * string) * string = (("a", "b"), "c")

This shows OCaml uses greedy tokenization for tuples. In particular, the parser will place ‘(‘ and ‘)’ around the longest set of alternating literals and commas.

Feedback:

Question **3**

Complete

12.00 points out of 12.00

Is it possible to write a compiler that translates C++ source code into JVM bytecode that can be executed by a Java interpreter, and implement C++ programs that way instead of the usual way? If so, give some pros and cons of building such a compiler; if not, explain why not.

JVM bytecode is platform agnostic, and this clashes with the fact that there is a lot of C++ code out there that is machine or platform dependent. Thus, translating would not be very feasible. As a more extreme case, you can embed assembly in your C++ code, which would definitely not translate to JVM bytecode if, say, a specific x86 instruction were being utilized, and you can call C functions like the GCC atomic operations that are specific to Intel chips. In general, C++ was designed with being able to program close to the platform/hardware in mind, whereas Java was intended to abstract away the platform, and promote cross-platform compatibility.

However, for many use cases for C++, code written is often not intended to be platform specific. In these cases, compiling to JVM bytecode is feasible, and any libraries native to a particular platform can be bound with something like JNA (Java Native Access). The pro of having a compiler to JVM bytecode would be portability between platforms, and, under certain circumstances, this could be possible. The con of such a compiler would be that platform specific code would have to be factored out and called through some API like JNA separately from the original source code.

Feedback:

Question **4**
Complete
12.00 points out of 12.00

Assuming you've solved Homework 1, write an OCaml function gsyms that accepts a grammar G in the style of Homework 1, and returns a list representing the set of nonterminal values of G, i.e., the values of any nonterminals appearing in any grammar rule of G. For example, the following expression should return true:

```
  equal_sets (gsyms awksub_grammar)
           [Expr; Lvalue; Incrop; Binop; Num]
```

As in Homework 1, your code may use only the Stdlib and List modules (though it may assume all the Homework 1 functions are implemented).

```
let gsyms grammar =
    let rec helper g = match g with
        | [] -> []
        | hd :: tl -> set_union [fst hd] (helper tl)
    in
    helper (snd grammar)

(* set_union was implemented earlier in HW1 *)
```

Feedback:

Question **5**
Complete
3.00 points out of 3.00

For what sort of grammar should gsyms return the empty list? Give an example of such a grammar and its corresponding language.

Returning an empty list would require a language without nonterminals. Such a language would not have any rules, and wouldn't be able to have tokens since the start symbol has to be a nonterminal. In essence, an empty language. Thus an example would be

```
let my_rules = [];;
let my_grammar = None, my_rules;;
equal_sets (gsyms my_grammar) [];;
```

Feedback:

Question **6**
Complete
13.00 points out of 15.00

Assuming you've both solved Homework 1 and implemented gsyms, write an OCaml function grecsyms that accepts a grammar G and returns a list representing the set of values of recursive nonterminals of G. A nonterminal symbol is "recursive" if the symbol is an ancestor of itself in a parse tree for one of the sentences of the language. For example, the following expression should return true:

```
equal_sets (grecsyms awksub_grammar)
        [Expr; Lvalue]
```

As in Homework 1, your code may use only the Stdlib and List modules (though it may assume Homework 1's functions and gsyms).

```
(* Only works partially in its current state - but it should extract Expr in the example at least *)

let grecsyms grammar =
    let rec recurse_thru_nalts rule nt = match rule with
        | [] -> []
        | N hd :: tl -> if hd = nt then [hd] else recurse_thru_nalts tl nt
                (* need to add something like @(find_rules_and_recurse hd (snd grammar)))
                but this would add infinite recursion, so need to keep track of rules encountered as well *)
        | T hd :: tl -> recurse_thru_nalts tl nt

    and recurse_thru_rules rules nt = match rules with
        | [] -> []
        | rules_hd :: rules_tl ->
            set_union (recurse_thru_nalts (snd rules_hd) nt) (recurse_thru_rules rules_tl nt)

    and find_rules_and_recurse nt grammar =
        let rules = List.filter (fun (n, rhs) -> n = nt) grammar in
        recurse_thru_rules rules nt

    and specific_nont nonts grammar = match nonts with
        | [] -> []
        | hd :: tl ->
            set_union (find_rules_and_recurse hd grammar) (specific_nont tl grammar)

    and nonts = gsyms grammar in
    specific_nont nonts (snd grammar)
```

Feedback:
As stated, only extracts directly recursive expr

Question **7**
Complete
5.00 points out of 5.00

For what sort of grammar should grecsyms return the empty list even though gsyms returns a nonempty list? Give an example of such a grammar and its corresponding language.

This would simply require a grammar with no recursive nonterminals. In other words, a language that is finite (recursion is what allows a language to be as large as it wants). For example:

```
let my_rules =
        [Expr, [T "one"; N Num; N Lvalue];
        [Lvalue, [N Num];
        [Num, [T "1"]]
let my_grammar = Expr, my_rules;
```

This language would only allow "one", "1", "1" and that's it.

Feedback:

Question **8**

Complete

18.00 points out of 18.00

Consider the hint code given for Homework 2, which defines the type ‘pattern’. Suppose we change that type’s definition by appending the following line to it:

| Diff of pattern * pattern

where Diff(P1, P2) is a pattern that matches any fragment that matches P1 but not P2; it is a set-difference operation on patterns. Modify the hint code to support this extension to Pattern. Minimize changes to the existing hint code.

For convenience, the Homework 2 hint code is given below.

```
(* DNA fragment analyzer.  *)

type nucleotide = A | C | G | T
type fragment = nucleotide list
type acceptor = fragment -> fragment option
type matcher = fragment -> acceptor -> fragment option

type pattern =
  | Frag of fragment
  | List of pattern list
  | Or of pattern list
  | Junk of int
  | Closure of pattern

let match_empty frag accept = accept frag

let match_nothing frag accept = None

let rec match_junk k frag accept =
  match accept frag with
    | None ->
        (if k = 0
         then None
         else match frag with
                | [] -> None
                | _::tail -> match_junk (k - 1) tail accept)
    | ok -> ok

let rec match_star matcher frag accept =
  match accept frag with
    | None ->
        matcher frag
                (fun frag1 ->
                   if frag == frag1
                   then None
                   else match_star matcher frag1 accept)
    | ok -> ok

let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None

let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)

let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher head) (mams tail)
  in mams ls

let rec make_or_matcher make_a_matcher = function
  | [] -> match_nothing
  | head::tail ->
      let head_matcher = make_a_matcher head
      and tail_matcher = make_or_matcher make_a_matcher tail
      in fun frag accept ->
           let ormatch = head_matcher frag accept
           in match ormatch with
                | None -> tail_matcher frag accept
                | _ -> ormatch

let rec make_matcher = function
  | Frag frag -> make_appended_matchers match_nucleotide frag
  | List pats -> make_appended_matchers make_matcher pats
  | Or pats -> make_or_matcher make_matcher pats
  | Junk k -> match_junk k
  | Closure pat -> match_star (make_matcher pat)
```

```
(* Modifications *)

type pattern =
  | Frag of fragment
  | List of pattern list
  | Or of pattern list
  | Junk of int
  | Closure of pattern
  | Diff of pattern * pattern        (* <- new *)

(* <- new function: similar to the implementation of make_or_matcher -> *)
let match_diff make_a_matcher p1 p2 =
    let p1_matcher = make_a_matcher p1
    and p2_matcher = make_a_matcher p2
    in fun frag accept ->
        let nomatch = p2_matcher frag accept
    in match nomatch with
        | None -> p1_matcher frag accept
        | _ -> None

let rec make_matcher = function
  | Frag frag -> make_appended_matchers match_nucleotide frag
  | List pats -> make_appended_matchers make_matcher pats
  | Or pats -> make_or_matcher make_matcher pats
  | Junk k -> match_junk k
  | Closure pat -> match_star (make_matcher pat)
  | Diff (p1, p2) -> match_diff make_matcher p1 p2    (* <- new *)
```

Feedback:

Question **9**
Complete
8.00 points out of 12.00

Would it be reasonable to change Homework 2 so that its solutions would support a similar set-difference operation on nonterminals? For example, this would let you specify a rule equivalent to "A statement can be an expression that is not an integer constant, followed by ';'." If so, explain how you'd go about changing Homework 2; if not, explain why not. You needn't *solve* the revised Homework 2; that is, your job here is to change the specification of the problem, not to solve the problem.

Similar to the previous problem, I'd reimplement HW2 with two matchers, one that checks for the patterns that we want, and one that checks for the patterns that we don't want. Matching a pattern that we don't want automatically invalidates, and we can match for a particular pattern we want like the original implementation.

Feedback:

Question **10**
Complete
14.00 points out of 15.00

You're a Java programmer who's a refugee from C++/C and really miss pointers. You want to tmplement a Java class 'Ptr' that sort of simulates C-style pointers into arrays. Here's the properties you want.

- Ptr objects are immutable: you cannot change them once created.
- Given a Java array A and an integer I where 0 <= I <= A.length, Ptr(A,I) creates a pointer to the Ith element of A. When I == A.length the pointer is one past the end of A. It is an error if A is null or I is out of range.
- Given a pointer P and an integer I, Ptr(P,I) creates a pointer representing P+I. This points to the Ith element after the element that P points to. It is an error if P is null or the resulting pointer is out of range for the array that P points to.
- Given a pointer P you can use P.get() to get the element it points to, and P.set(V) to set the element's value. It is an error if P is null or points just past the end of the array.
- Given two pointers P, Q to the same type T, you can compare them for equality with P.equals(Q). This returns true if and only if they point to the same element of the same object, or if they are both null.
- Given two pointers P, Q to the same type T, you can compare them for less-than with P.lt(Q). This returns true if and only if P precedes Q in the underlying object. It is an error if P and Q point to different objects, or if P or Q are null.

Write the code for the class Ptr. Use generics when possible. Assume the existence of a method 'error()' that your code should call whenever there is an error. (In real Java you'd use exceptions but we haven't covered exceptions yet.)

```
public class Ptr<T> {
        protected T[] myList;
        protected int my_i;

        private void check_error() {
                if ((myList == null) || (i >= myList.length)) {
                        error();
                }
        }

    public Ptr(T[] A, int i) {
                myList = A;
                my_i = i;
               check_error();
    }

    public Ptr(Ptr<T> P, int i) {
        myList = P.myList;
        my_i = P.my_i + i;
        check_error();
    }

    public T get() {
        check_error();
        return myList[my_i];
    }

    public void set(T v) {
        check_error();
        myList[my_i] = v;
    }

    public boolean equals(Ptr<T> Q) {
        if ((myList == null) && (Q.myList == null)) {
                return true;
        }
                if ((myList == Q.myList) && (my_i == Q.my_i)) {
                return true;
        }
        else {
```

Feedback:
check if i < 0

should use private, coudl still be modified within package

Question **11**
Complete
8.00 points out of 10.00

Explain any shortcomings in your Ptr code. Or if there aren't any significant shortcomings, compare how well Ptr works in Java, compared to how well similar code would work in C++.

Potential shortcomings: the code may have some syntactic issues, but generics are used where appropriate, error checking is implemented, all required methods are implemented, and "protected" keywords are used to ensure immutability.

Given that this Ptr code works, but it's much less efficient than C++ code. The Java code requires instantiation of an object for each pointer, which can be slow and resource intensive, and we have to rely on garbage collection to dispose of pointers. If pointers were to be used to the same extent as they are in C++, there would also be huge performance issues in terms of having to access the object and then the list itself as well as running functions instead of doing pointer arithmetic.

Feedback:
What about differences with ergonomics, safety, and handling nulls?

Question **12**
Complete
2.00 points out of 2.00

In C, the type 'void *' is a generic pointer: although you cannot do anything with it directly (other than comparing it for equality), you can assign a value of any unqualified pointer type to a variable of type 'void *', and conversely you can assign a value of type 'void *' to a variable of any unqualified pointer type. If a pointed-to type has a qualifier, e.g., 'char const *', you can convert between it and 'void const *' instead; you can also assign a 'void *' value to a 'char const *' variable.

Use 'void *' to show that C is not a strongly-typed language.

```
int main() {
   int i = 65;
   void *vp = &i;
   printf("%d %c", *((int*)vp), *((char *)vp));
}
```

Prints "65 A" from a void pointer. Notably through 'void *' we can treat an int as a char or an int (or something else). Casting is a design philosophy that makes C inherently not strongly-typed. From the short example, it's easy to see how we "cheated" the types, and subverted the type-checking mechanism.

Feedback:

Question **13**
Complete
6.00 points out of 8.00

Discuss how subtyping works in C when pointer-to-void is allowed. For example, is ‘void const *’ a subtype of ‘char *’?

In the case of ‘void const *’ and ‘char *’, ‘void const *’ encompasses a larger set of values than ‘char *’, and it has a smaller set of operations. In general, subtypes are a subset of values with a superset of operations. Thus, it might be more appropriate to say that pointer-to-void is a “super”-type and all other types of pointers are subtypes, so in the case of the example given, ‘void const *’ is the parent type and ‘char *’ is the child subtype. This is made even more apparent by the fact that you can cast a pointer-to-void to the other subtypes.

Feedback:
Good reasoning.

this compiles without errors, only warnings, showing that the subtyping relation doesn't really work neatly

```
char* foo = "hello";
void const* bar = foo;
char* bat = bar;
```

in a subtyping relation, you should only be able to assign one way, not the other. So the assignment operation breaks the subtyping relation.

◄ Sample obsolete midterm ...

Jump to...