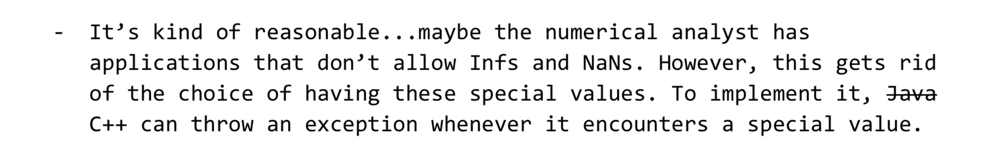
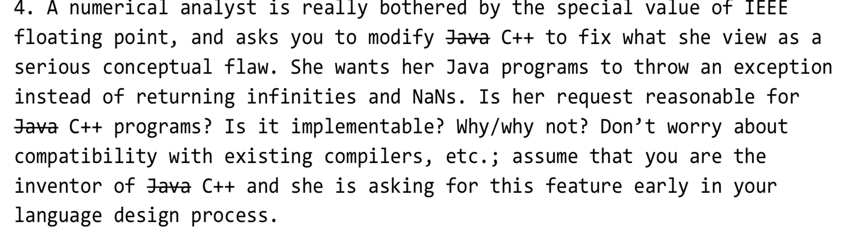
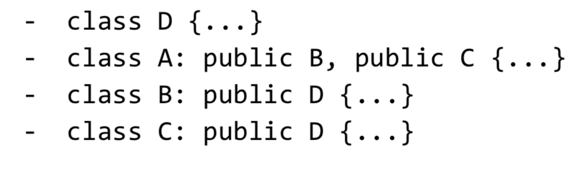
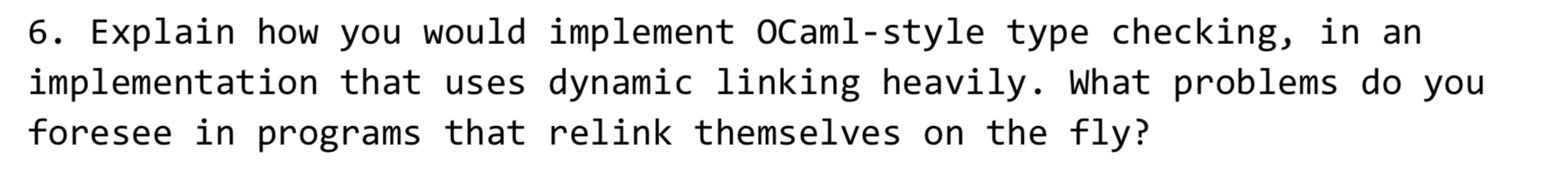
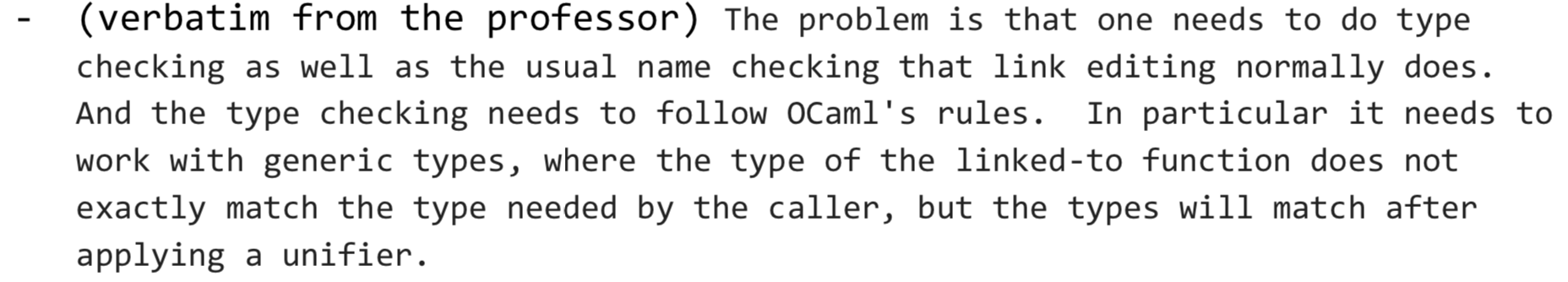
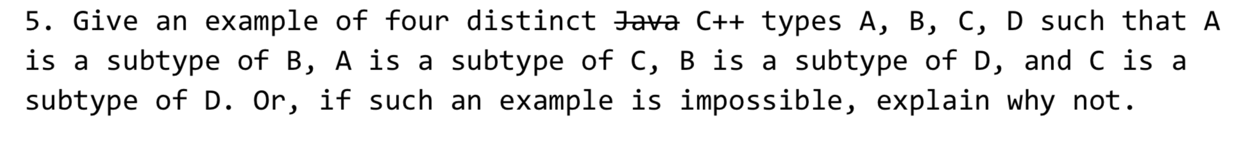
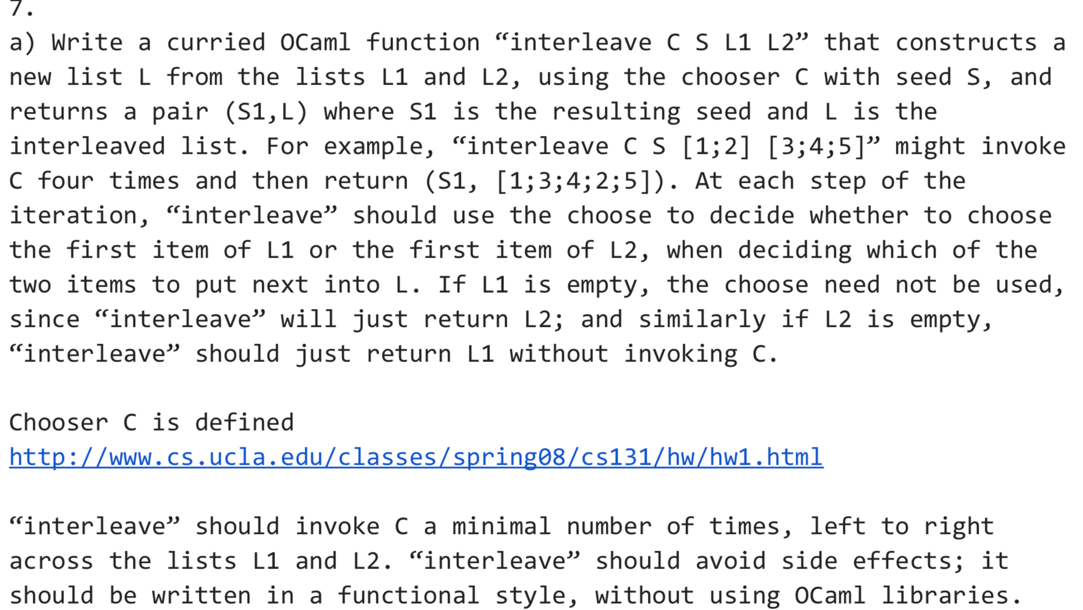
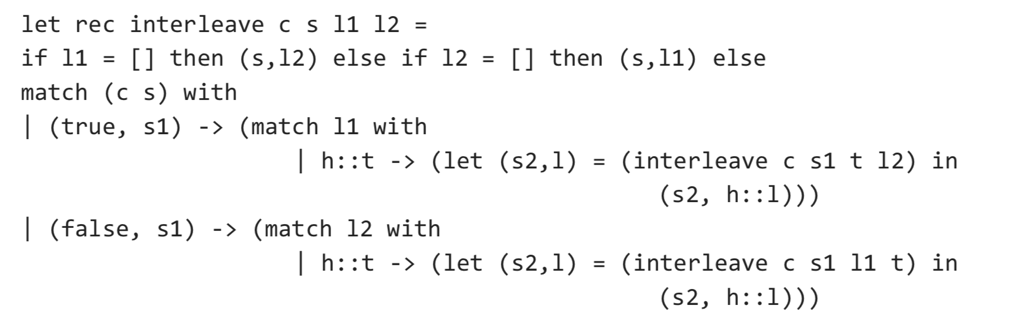
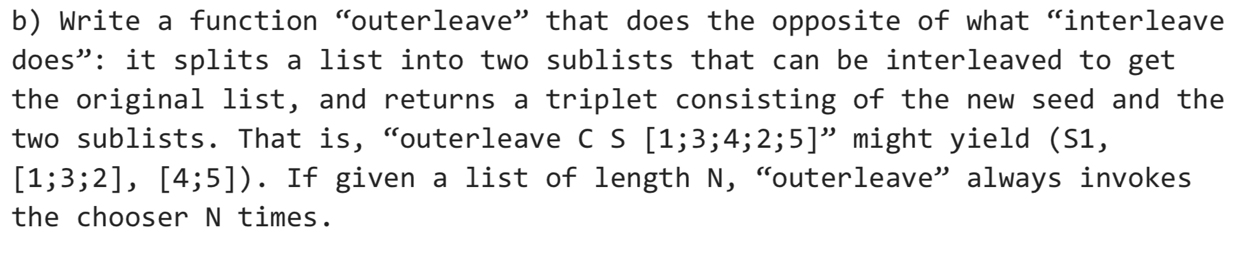
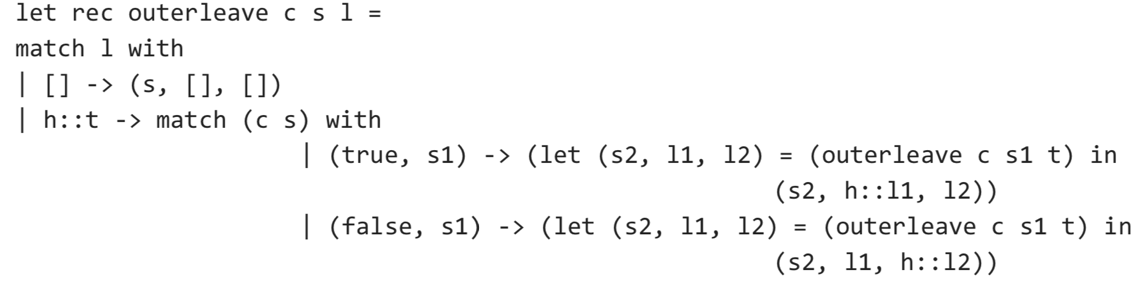
|  |  |  |  |
| --- | --- | --- | --- |
| **Lecture 4**  Languages with dynamic grammars  Prolog   * + - * Predefined operators         + `:op(500, yfx, [+, x])         + 500 gives the priority         + yfx specifies whether the operator is unary or binary, and what the operands are   x and y are its operands  y allows equal priority  x means that only operators with higher priority (that is, with lower priority number, here lower than 500).  The operand is left-associative, because operands of equal priority is expanded on the left.   * + - * + :-op(400, fyx, [\*,/])         + :-op(200, fx, [+, -])   Unary + and -   * + - * + :-op(700, xfx, [==, ==...])   Lower priority than others  Operands must have higher priority  So 3 = 4 = 5 is illegal  In C, a == b == c   * + - * + :-op(300, xfy, [\*\*])   Right-associative because the operator with equal priority is on the right.   * + - * Extra constraints are expressed non-grammatically | | **Homework 1** | |
| let rec subset a b =  match a with  | [] -> true  | h :: t ->  if List.mem h b then  subset t b else false;; | **List.mem** – check to see if an element is a member of a list  **List.sort** – well, what do you think. |
| let rec uniq d e =  match d with  | [] -> e  | h :: t ->  if not(mem h e) then  uniq t (h::e)  else  uniq t e | let set\_union a b =  List.sort compare (uniq (uniq a []) (uniq b [])) ;; |
| let rec compute eq f p x e =  if (=) (List.length e) p  then List.hd e else  compute eq f p (f x) ((f x)::e) ;;  let rec computed\_periodic\_point eq f p x =  match p with  | 0 -> x  | \_ ->  if eq x (compute eq f p x []) then x else computed\_periodic\_point eq f p (f x) ;; | let rec computed\_fixed\_point eq f x =  if eq x (f x) then x else computed\_fixed\_point eq f (f x) ;;  **List.reverse** – again, what do you think. |
| **The blind alley (10)**  let filter\_blind\_alleys g = ((fst g),(List.rev( reverser (uniq (((runthrough app (snd g) (keyword (first\_run (snd g) []) []) (first\_run (snd g) [])))) []) (snd g) [])));; | |
| (\*------1-------\*)  (\* basic type definition \*)  type ('nonterminal, 'terminal) symbol =  | N of 'nonterminal  | T of 'terminal;; | (\*------3-------\*)  (\*takes in list of rules - rs, which should be snd g - then outputs the initial 'goodList', a list of approved rules\*)  let rec first\_run rs goodList =  match rs with  | [] -> goodList  | h::t ->  if List.for\_all term (snd h) then first\_run t (h::goodList)  else first\_run t goodList;; | (\*------6-------\*)  (\*pass in the rhs of a rule, to check if each member of rhs is valid. Output true if total rule is valid\*)  let rec validrule symblist goodKey =  match symblist with  | [] -> true  | h::t -> if (non h goodKey) then true &&  (validrule t goodKey) else false;; | (\*------8-------\*)  let rec runthrough app rs goodKey goodList =  if equal\_sets goodList (app rs goodKey goodList) then goodList  else runthrough app rs (keyword (app rs goodKey goodList) goodKey) (app rs goodKey goodList);; |
| (\*------2-------\*)  (\* meant to handle rhs for basic run of terminal types - takes in a symbol, outputs true if terminal, false if nonterminal\*)  let term symb =  match symb with  | T (\_) -> true  | N (\_) -> false ;; |
| (\*------7-------\*)  (\*Check if a rule is valid, if so add its rhs to goodKey, add the rule itself to goodList\*)  let rec app rs goodKey goodList =  match rs with  | [] -> goodList  | h::t -> if validrule (snd h) goodKey then app t ((N (fst h))::goodKey) (h::goodList)  else app t goodKey goodList;; | (\*------9-------\*)  let rec reverser goodList original sorted =  match original with  | [] -> sorted  | h::t ->  if List.mem h goodList then reverser goodList t (h::sorted)  else reverser goodList t sorted;; |
| (\*------4-------\*)  (\*takes in a list of rules - again, snd g - , then outputs an initial 'goodKey', a list of lhs. Meant to be run on goodList\*)  let rec keyword rs goodKey =  match rs with  | [] -> goodKey  | h::t -> keyword t ((N (fst h))::goodKey);; |
| (\*------5-------\*)  let non symb goodKey =  if (List.mem symb goodKey || term symb) then true else false;; |

|  |  |
| --- | --- |
| **Homework 2**  Part1: Grammar conversion  (\*meant to grab an alternate list for 1 nt\*)  let rec altlist nt rules alt =  match rules with  | [] -> alt  | h::t ->  if (=) (fst h) nt then  altlist nt t ((snd h):: alt)  else altlist nt t alt ;;  (\*the one with the correct order\*)  let rec convert\_grammar gram1 =  match gram1 with  | (ss, rules) -> (ss, function nt -> (List.rev (altlist nt rules [])));; | let parse\_prefix gram accept frag =  let rec match\_element rules rule accept derivation frag = match  rule with  | [] -> accept derivation frag  | \_ -> match frag with  | [] -> None  | curr\_prefix::r\_frag -> match rule with  | [] -> None  | (T term)::rhs -> if curr\_prefix = term then  (match\_element rules rhs accept derivation r\_frag) else  None  | (N nterm)::rhs -> (matcher nterm rules (rules nterm)  (match\_element rules rhs accept) derivation frag)  and matcher start rules matching\_start\_rules accept derivation frag =  match matching\_start\_rules with  | [] -> None  | top\_rule::other\_rules -> match (match\_element rules top\_rule  accept (derivation@[start, top\_rule]) frag) with  | None -> matcher start rules other\_rules accept derivation  frag  | Some res -> Some res in  matcher (fst gram) (snd gram) ((snd gram) (fst gram)) accept [] frag |

|  |  |  |
| --- | --- | --- |
| 1.“Ireland has leprechauns galore.” is an example of a particular kind of syntactic construct in English. Can you construct a similar example in C++, OCaml, or Java? If so, give an example: if not, explain why not.  Artificial languages are generally more carefully-designed than natural languages. Thus, these kinds of exceptions should be rare, if they exist at all.  2. a) Write an OCaml function ‘twice’ that accepts a function f and returns a function g such that g(x) equals f(f(x)). For simplicity’s sake you can assume that f is free of side effects, and you can impose other restrictions on f and x. Try to keep the restrictions as minor as possible and explain any restrictions you impose. Or, if “twice” cannot be written easily in OCaml, explain why not.  let twice f x = f(f(x))  (restriction is that f is ‘a -> ‘a  b) Same as a) except write a function ‘half’ that accepts a function f and returns a function g such that f(x) equals g(g(x))   * It is difficult, since its implementation must depend on the specific function f used. * For example,if f is identity function f(x)=x,then g is easy—also the identity function. * However,if f is the sine function, then g is super difficult--given x,g(g(x)) = sin(x)??? * The restriction, like a), is that f is‘a->’a * c) Give the types of ‘twice’ and ‘half’ - twice:(‘a->’a)->‘a->‘a - half:(‘a->‘a)->‘a->‘a | | |
| 3. Consider the following grammar for a subset of the C++language.  expression: expression ? expression : expression expression != expression expression + expression  ! expression INTEGER-CONSTANT (expression)  For example,(!!0+1!=2?3:4) is read as “if not-not-0 plus 1 does not equal 2, then 3 else 4, and evaluates to 4. | a) What are the tokens of this subset of C++?  ? :! = + ! INTEGER\_CONSTANT ()  b)Show that this grammar is ambiguous  Draw the two parse trees for expression 1+2+3  c)Rewrite the grammar so that it is no longer ambiguous, resolving any ambiguities in the same way that c++ does. Recall that in C++, the expression  (0!=1!=2||3+!4+5||6?7:8?9:10) is like (((((0!=1)!=2)||((3+(!4))+5))||6)?7:(8?9:10))  d)Translate the rewritten grammar into a syntax diagram  eg. E2  o------> E3 ----->o  |<--!=<--| | E->E2?E2:E|E2  E2->E2!=E3|E3  E3->E3+E4|E4  E4->!E4|E5  E5->INTEGER-CONST|(E) |







output a function that is a parser; it will take a list of tokens as input, and then output a parse in the form of either parse tree or derivation. Real compilers will do a lot more, including but not limited to type checking, identifier checking, and generating machine code. The problem itself is generally hard and please use the simplest algorithm you can think of. You don't have to worry too much about the efficiency of your algorithm.

