1 Printing and parsing

The Printf and Scanf modules in the OCaml standard library expose functions for printing and parsing using format strings. For example, here is a call to sprintf to build a string from a pair of integers:

```
# Printf.sprintf "(%d, %d)" 3 4;;
- : string = "(3, 4)"
```

And here is a call to sscanf to read a pair of integers from a string:

```
# Scanf.sscanf "(3, 4)" "(%d, %d)" (fun x y -> (x, y));;
-: int * int = (3, 4)
```

The full implementations of Printf and Scanf consist of many thousands of lines of code. (See the files camlinternalFormat.ml, camlinternalFormatBasics.ml, and scanf.ml in the OCaml distribution.) This exercise focuses on cut-down variants of the functions consisting of only a few lines.

The Caml implementation processes a format string such as 1(7,%)" in two steps. First, the Sirguille and by the Campiler and the AST by ascribing the type _ format to the string at the OCaml prompt:

Second, when the program runs, the sprintf and sscanf functions interpret the AST to determine how to turn values into strings or to read values from strings.

This question focuses on the second step, and so we'll write ASTs directly rather than using format strings. Our goal is to build staged versions of sprintf and sscanf that avoid interpretative overhead by generating code specialized to the format strings.

The file exercise3_printf.ml contains an implementation of format specifiers as a data type:

```
type (_,_) fmt =
  | Int : (int -> 'a, 'a) fmt
  | Lit : string -> ('a, 'a) fmt
  | Bool : (bool -> 'a, 'a) fmt
  | Cat : ('a, 'b) fmt * ('b, 'c) fmt -> ('a, 'c) fmt
let (%) x y = Cat (x, y)
```

and an implementation of a function sprintf that interprets format specifiers to turn values into strings:

```
val sprintf : ('a, string) fmt -> 'a
```

For example, here is a call to sprintf equivalent to the call to the standard library function on page 3:

```
# sprintf (Lit "(" % Int % Lit ", " % Int % Lit ")") 3 4;;
- : string = "(3, 4)"
```

(a) Build a staged version of sprintf with the following type:

```
val sprintf_staged : ('a, string) fmt -> 'a code
```

that generates code to print a string rather than printing the string directly:

```
# sprintf_staged (Lit "(" % Int % Lit ", " % Int % Lit ")");;
- : (int -> int -> string) code =
.< fun i ->
fun i ->
```

Assignment Project Exam Help ("")">

The goal is to eliminate interpretative overhead, and so the generated code should not contain any values of the fmt type (Int, Lit, etc.).

https://powcoder.com

(b) Following the idealized staging process for sprintf results in less than optimal code. For example, adjacent literals in a format specifier result in a call to the string torical partiest operator in the generated code, even though the information needed for concatenation is available our ingreded generation:

```
# sprintf_staged (Lit "a" % Lit "b");;
- : string code = .<"a" ^ "b">.
```

As is often the case, the problem can be solved by using partially-static data.

Give an implementation of the following module for partially-static strings:

```
module Ps_string : sig
  type t
  (** The type of partially-static strings *)

val sta : string -> t
  (** Build a partially-static string from a static string *)

val dyn : string code -> t
  (** Build a partially-static string from a dynamic string *)

val cd : t -> string code
  (** Turn a partially-static string into a dynamic string *)

val (++) : t -> t -> t
  (** Concatenate two partially-static strings *)
end
```

that eliminates as much concatenation as possible from the code returned by cd:

(c) Use your implementation of Ps_string to build an improved version of sprintf_staged that concatenates adjacent literal strings during code generation:

```
# sprintf_improved (Lit "a" % Lit "b");;
- : string code = .<"ab">.
```

(d) A second way to improve generated code is to take advantage of the fact that the bool type has only two inhabitants. A straightforward implementation of sprintf_staged might generate the following code for printing booleans

Alowever with branchest op he possible to Evalues we during equivalent code without any concatenations.

```
https://powcodersecom

if c then "(true, true)"

to then "(true, true)"

if c then "(false, true)"

else "(false, false)"

Add WeChat powcoder
```

Implement the following *if-insertion* interface (analogous to the *let-insertion* discussed in the lectures):

```
val split : bool code -> bool
val if_locus : (unit -> 'a code) -> 'a code
```

and use it to give a third implementation of the staged printf function:

```
val sprintf_improved2 : ('a, string) fmt -> 'a code
```

that generates concatenation-free code for the format specifier above.

(NB: the continuations bound in effect handlers can only be invoked once. The function Obj.clone, that builds a copy of a continuation, can be used to work around this limitation.)

(e) The format specifiers used for printing with sprintf can also be used for parsing. Give an implementation of the parsing function sscanf with the following type:

```
val sscanf : ('a, 'b) fmt -> string -> 'a -> 'b
```

using the supplied helper functions read_exact, read_int, and read_bool, or otherwise.

(f) Give a staged implementation of sscanf that accepts a format specifier and returns code for a parsing function:

```
val sscanf_staged ('a, 'r) fmt -> ('string -> 'a -> 'r) code =
```

For example, your function might generate code like this for parsing an integer between parentheses:

Assignment Project Exam Help

As with sprintf_staged, the goal is to eliminate interpretative overhead, and so the generated code should not contain any values of the fmt type (Int, Lit, etc.).

(g) The code in part (f) powcoder com implementation, due to the nested let bindings and unnecessary intermediate tuples.

Add WeChat powcoder

Let insertion can help generate code that is more idiomatic, and probably more efficient. However, the implementation of let insertion presented in lectures only builds variable bindings (let x = e in ...), not tuple bindings (let (x, y) = e in ...).

Implement the following interface for let insertion with tuple bindings:

```
val genlet2 : ('a * 'b) code -> ('a code * 'b code)
val let_locus2 : (unit -> 'a code) -> 'a code
```

and use it to give a second implementation of sscanf_staged that generates code without nested let bindings or superfluous tuple values:

```
# sscanf_staged_improved (Lit "(" % Int % Lit ")");;
- : (string -> (int -> '_a) -> '_a) code =
.< fun s ->
    fun k ->
    let (x<sub>1</sub>,s) = read_exact "(" s in
    let (x<sub>2</sub>,s) = read_int s in
    let (x<sub>3</sub>,s) = read_exact ")" s in
    k x<sub>2</sub>>.
```

(20 marks)

2 Staged searching

Multi-stage programming is a good fit for algorithms that involve a preparatory step, since the preparation can typically be performed during code generation to build specialized code. For example, the performance of string searching can be improved by building a table with information that avoids more comparisons than necessary.

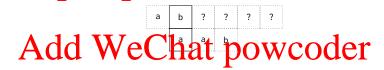
As a concrete example, consider the task of searching for a string aab (the "needle") within a second string (the "haystack"). A naive implementation might proceed as follows:

First, compare aab with the first three characters ab? (→ failure!)

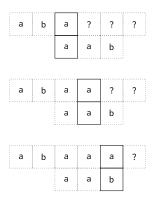


Assignment Project Exam Help

Next, compare as with the second three characters b?? (\sim failure!) POWCOGET.COM

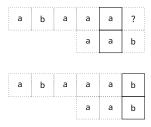


Next, compare aab with the third three characters aaa (→ failure!)



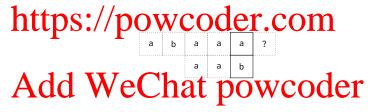
Finally, compare aab with the fourth three characters aab (→ success!)



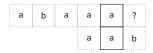


This approach is obviously correct, but it involves unnecessary work, repeatedly examining the same characters. For example, the fourth comparison starts by examining the fourth and fifth characters of the haystack, even though the previous comparison has already determined that those character are both a. The efficiency of the search strategy can be improved by using the information revealed by each comparison to determine where to restart the search.

Information about where to restart after failure depends only on the needle, not the haystack. If the comparison fails at the ith character of the needle, then the previous i-1 characters of the haystack must match the first i characters of the needle. For example Soft period of the haystack must comparison illustrated above), then the previous two characters of the haystack must both be a



and so the search can skip the first character of the next search, and begin on the second:



Similar reasoning for each character of the needle leads to the following table:

where the entry for the ith character indicates that, if a comparison fails after i characters, the first i - table[i] characters of the next comparison can be skipped.

More precisely, each new comparison should start after the portion of the haystack just examined, unless there is some overlap between the end of the successfully matched portion and the beginning of the needle. If the i most recent characters examined in the haystack correspond to the i characters at the beginning of the needle then the new comparison should begin with the beginning of the needle moved back i characters from the failure point; however, the characters in the overlapping portion can be skipped.

In the example above, at most one character can be skipped. Longer needles typically allow skipping more characters. For example, here is the table for abcdabd:

If the comparison fails after successfully matching abc

Assignment Project Exam Help

then the next comparison can skip forward three (3 - table[needle[3]]) characters, since neither attraction we will be comparable to the column.



(a) Implement the naive search functions

```
val naive_search_from : string -> string -> int -> bool
val naive_search : string -> string -> bool
```

so that naive_search_from n h i compares n with each length(n) substring of h, starting from index i in h, and using character-wise comparison, and so that naive_search simply calls naive_search_from, passing 0 as the third argument.

(b) Following the idealized staging process described in the lectures, implement a staged version of naive_search that accepts the needle string and builds code for a search function specialized to the needle:

```
val naive_search_staged : string -> (string -> bool) code
```

For example, your implementation might generate code like the following for the needle aab:

```
# naive_search_staged "aab";;
- : (string -> bool) code =
.< fun s ->
```

```
let rec loop i =
  ((s.[i] = 'a' && s.[i+1] = 'a' && s.[i+2] = 'b')
  || loop (i + 1) in
try loop 0 with Invalid_argument _ -> false >.
```

(c) (i) Implement the functions

```
val proper_prefixes : string -> string list
val proper_suffixes : string -> string list
```

that respectively compute the proper prefixes and proper suffixes of a string. A proper prefix or suffix of s is a prefix or suffix of s that is shorter than s. For example,

```
# proper_prefixes "abcd";;
- : string list = ["abc"; "ab"; "a"]
# proper_suffixes "abcd";;
- : string list = ["bcd"; "cd"; "d"]
```

As significant proper project and proper suffixes or otherwise, implement a Help

```
val skip_table : string -> (char * int) list
```

The still past that the length of the longest proper prefix of patto..i] that is also a proper suffix of the same string.

(d) Using sking taking from part (c) (ii) implement and stage the improved search algorithm as a function of the following type:

```
val fancy_search_staged : string -> (string -> bool) code
```

For example, your function might generate code like the following for the needle abab:

```
.< fun s ->
    let rec match_b i =
          if s.[i + 3] = 'b'
          then true
          else match_bab (i + 2)
    and match_ab i =
          if s.[i + 2] = 'a'
          then match_b i
          else match_abab (i + 2)
    and match_bab i =
          if s.[i + 1] = 'b'
          then match_ab i
          else match_abab (i + 1)
    and match_abab i =
          if s.[i] = 'a'
          then match_bab i
          else match_abab (i + 1) in
   try match_abab 0 with Invalid_argument _ -> false >.
```

Note that when match_b fails, it restarts the search by calling match_ab, not match_abab. And note, too, that the failure case in match_b and match_ab bumps the index by 2 to avoid unnecessarily re-scanning parts of the haystack already seen.

Although the example code above uses let rec, you may need to use an encoding of recursive functions, such as the reference-based approach discussed in Lecture 15, since generating let rec binding groups of arbitrary size is difficult or impossible. (However, solutions that use let rec are welcome, if you can find an approach that works.)

(15 marks)

Assignment Project Exam Help https://powcoder.com Add WeChat powcoder