Programming Languages
CSCI-GA.2110.001 Fall 2021

Homework 1

ANSWERS

You should write the answers using Word, latex, etc., and upload them as a PDF document. No implementation is required. Since there are drawings, if you prefer, you can hand write the answers and scan them to a PDF.

1. Provide regular expressions for defining the syntax of the following. You can only use concatenation, |, $*$, ϵ (the empty string), parentheses, and expressions of the form [A-Z], [0-3], [A-Z0-3], etc., to create regular expressions. That is, you cannot use + or any kind of count indicator.

   (a) Strings consisting of any number of lower case letters, upper case letters, and digits, such that the occurrences of upper case letters and digits are alternating. That is, there cannot be two upper case letters in the string without a digit somewhere in between them and there cannot be two digits in the string without an upper case letter somewhere between them. For example, "aAbc3dEf4ghIjk5m" would be an example of such a string. However, "aAbc3dEfGhi4jk" would not be a valid string, since there is no digit between E and G.

   Answer (there can be other correct answers):
   `[a-z]*(([A-Z][a-z]*[0-9][a-z]*)*([A-Z]|ϵ) | ([0-9][a-z]*[A-Z][a-z]*)*([0-9]|ϵ))[a-z]*`

   (b) Positive and negative floating point literals that specify a positive or negative exponent, such as the following: 243.876E11 (representing 243.867×10^{11}) and -135.24E-4 (representing -135.24×10^{-4}). There must be at least one digit before the decimal point and one digit after the decimal point (before the "E").

   Answer:
   `(+|-|ϵ)[0-9][0-9]*.[0-9][0-9]*E(+|-|ϵ)[0-9][0-9]*`
   You can add additional restrictions, e.g. no leading zeros, if you like, but it wasn't required.

   (c) Variable names that (1) must start with a letter, (2) may contain any number of letters, digits, and _ (underscores), and (3) may not contain two consecutive underscores.

   Answer:
   `[A-Za-z][A-Za-z0-9]*(_[A-Za-z0-9][A-Za-z0-9]*)*(_ | ϵ)`

2. (a) Provide a simple context-free grammar for the language in which the following program is written. You can assume that the syntax of names and numbers are already defined using regular expressions (i.e. you don't have to define the syntax for names and numbers). In this language, a program consists of a sequence of function definitions.

```
fun fac 0 = 1
 |  fac n = n * fac(n-1)
```

```
fun foo x y =
   let val z = x + y
       fun bar n = n * 2
   in bar z
   end
```

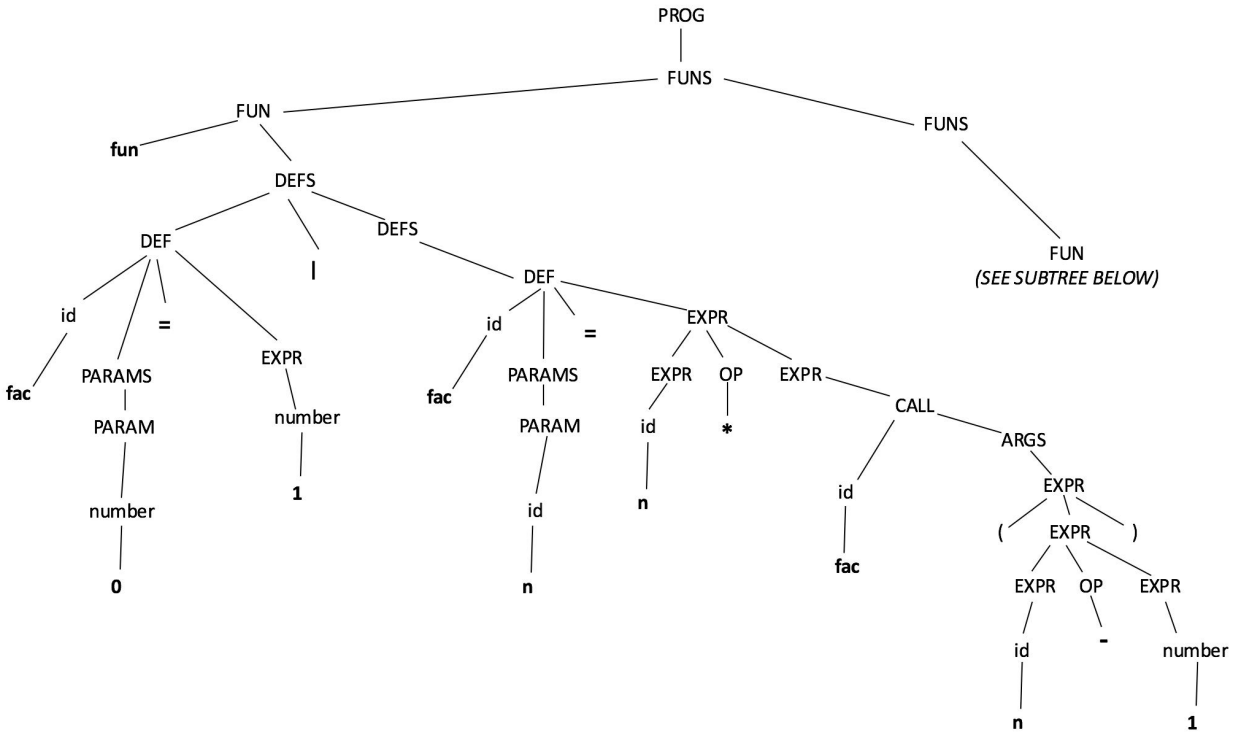
You only have to create grammar rules that are sufficient to parse the above program. Your starting non-terminal should be called PROG (for "program") and the above program should be able to be derived from PROG.

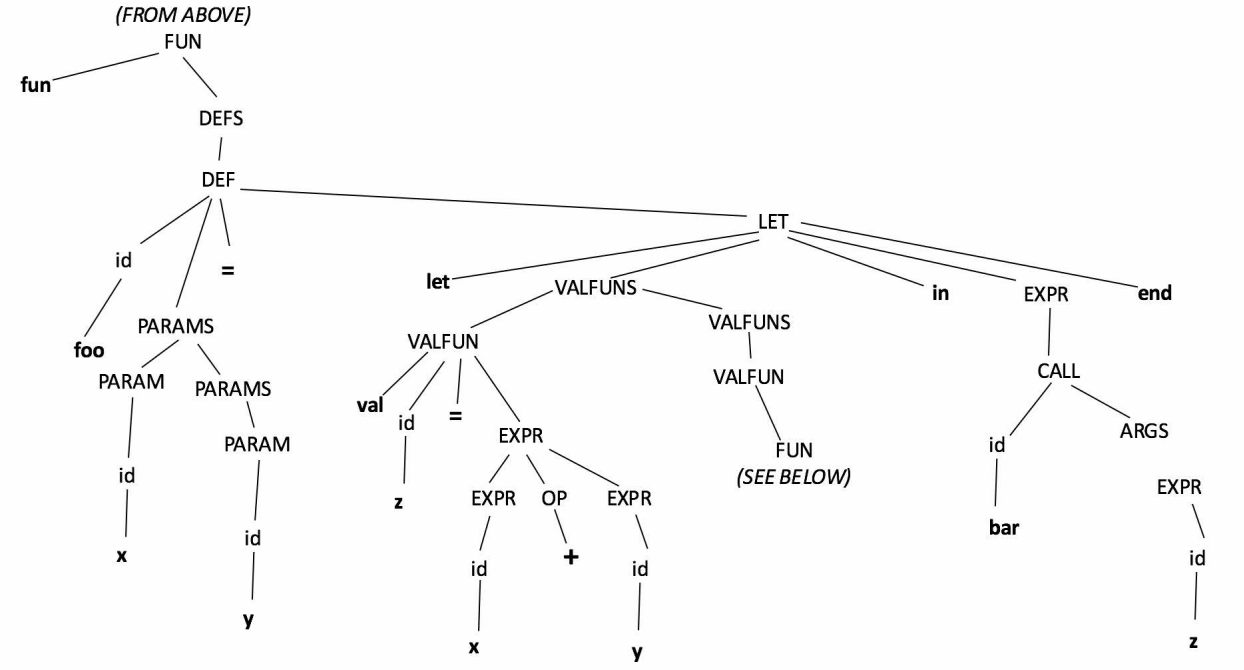
Here is one such grammar - though other ones are certainly possible. Non-terminals are in upper case letters.

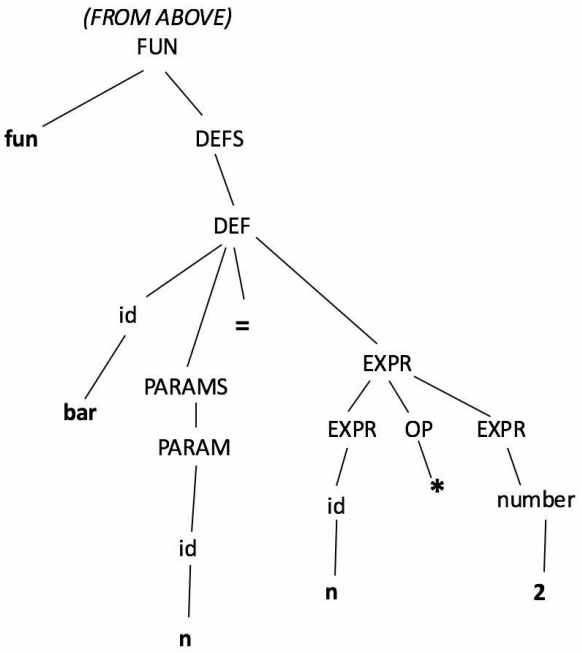
```
PROG → FUNS
FUNS → FUN | FUN FUNS
FUN → fun DEFS
DEFS → DEF | DEF "|" DEFS
DEF → id PARAMS = EXPR
PARAMS → PARAM | PARAM PARAMS
PARAM → id | number
EXPR → id | number | ( EXPR ) | EXPR OP EXPR | CALL | LET
OP → * | +
CALL → id ARGS
ARGS → EXPR | EXPR ARGS
LET → let VALFUNS in EXPR end
VALFUNS → VALFUN | VALFUN VALFUNS
VALFUN → val id = EXPR | FUN
```

Note that the above grammar is ambiguous, which can be fixed by modifying the production for EXPR (and probably ARGS), as discussed in class (but not required for this assignment). Also, the quotes around the use of | in the DEFS rule indicate that it is a terminal in the grammar, rather than being used to separate the different possibilities.

(b) Draw the parse tree for the above program, based on a derivation using your grammar.


PROG
FUNS
FUN
fun
DEFS
DEF
id
fac
=
PARAMS
PARAM
number
0
EXPR
number
1
|
DEFS
DEF
id
fac
=
PARAMS
PARAM
id
n
EXPR
EXPR
id
n
OP
*
EXPR
CALL
id
fac
ARGS
EXPR
(
EXPR
)
EXPR
OP
EXPR
id
n
-
number
1
FUNS
FUN
(SEE SUBTREE BELOW)


(FROM ABOVE)
FUN
fun
DEFS
DEF
id
foo
=
PARAMS
PARAM
id
x
PARAMS
PARAM
id
y
LET
let
VALFUNS
VALFUN
val
id
z
=
EXPR
EXPR
id
x
OP
+
EXPR
id
y
VALFUNS
VALFUN
FUN
(SEE BELOW)
in
EXPR
CALL
id
bar
ARGS
EXPR
id
z
end



Your parse tree does not need to show each actual identifier (`fac`, `n`, `foo`, etc.) or number, but rather the corresponding leaves can just be `id` or `number`.

3. (a) Define the terms *static scoping* and *dynamic scoping*.

In static scoping, the body of a function is evaluated in the environment of the function definition. In dynamic scoping, the body of a function is evaluated in the environment of the function call.

(b) Give a simple example, in any language you like (actual or imaginary), that would illustrate the difference between static and dynamic scoping. That is, write a short piece of code whose result would be different depending on whether static or dynamic scoping was used.

Answer (just one of an infinite set of possibilities):

```
procedure A()
  x: integer = 5;

 procedure B()
 begin
  print(x); (* static scoping prints 5, dynamic prints 10 *)
 end;

 procedure C()
   x: integer = 10;
 begin
   B();
 end;
```

```
begin (*A*)
  C();
end;
```

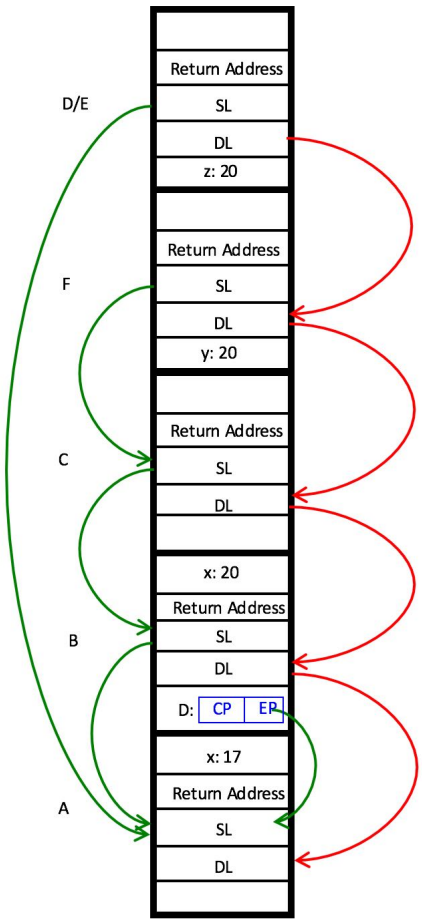
(c) Why do all modern programming languages use static scoping? That is, what is the advantage of static scoping over dynamic scoping?

In static scoping, the behavior of a function – including all the non-local variables it references – can be understood from looking at its definition, possibly including the surrounding scopes. In dynamic scoping, each call to a function can have a different behavior, due to the fact that different non-local variables can be referenced within the function, based on the variables that are visible at the site of the function call. Thus, to understand what a function does, it may be necessary to look at every place that it is called – which in a large program could be many, many places.

4. (a) Draw the call stack for the following program that would exist when the print statement is executed. Assume the language is statically scoped. Show at least the local variables and parameters (including any closure), the static and dynamic links, and identify which stack frame is for which procedure.

```
procedure A()
   x: integer := 17;

   procedure B(procedure D)
      x: integer := 20;

      procedure C()
        procedure F(y:integer)
        begin (* F *)
           D(y)
        end; (* F *)
      begin (* C *)
         F(x);
      end; (* C *)

   begin (* B *)
      C();
   end (* B *)

   procedure  E(z:integer)
   begin (* E *)
      print(z, x);
   end; (* E *)

begin (* A *)
  B(E);
end; (* A *)
```

Answer:



(b) What would the program print?

Answer:

```
20 17
```

(c) Suppose, instead, that the language was dynamically scoped. What would the program print?

Answer:

```
20 20
```

5. For each of these parameter passing mechanisms,

   (a) pass by value

   (b) pass by reference

   (c) pass by value-result (assume the address of an argument is computed only once)

   (d) pass by name

   state what the following program (in some Pascal-like language) would print if that parameter passing mechanism was used:

```
program foo;
  var i,j: integer;
```

```
    a: array[1..10] of integer;

  procedure f(x,y:integer)
  begin
    x := x * 4;
    i := i + 1;
    y := a[i] * 5;
  end

begin
  for j := 1 to 10 do a[j] = j;
  i := 2;
  f(i,a[i]);
  for j := 1 to 10 do print(a[j]);
end.
```

Answer:
Pass by Value: 1 2 3 4 5 6 7 8 9 10
Pass by Reference: 1 45 3 4 5 6 7 8 9 10
Pass by Value Result: 1 15 3 4 5 6 7 8 9 10
Pass by Name: 1 2 3 4 5 6 7 8 45 10

6. In Ada, define a procedure containing two tasks such that (1) the first task prints the odd numbers from 1 to 99 in order, (2) the second task prints the even numbers from 2 to 100 in order, and (3) for each odd number N, the value of N and N+1 are printed concurrently but there is no other concurrent printing of numbers. For example, 1 and 2 are printed concurrently, and 3 and 4 are printed concurrently, but 1 and 2 must be printed before 3 and 4.

Answer:

```
procedure Hw is

 task One;
 task Two is
    entry Go;
 end Two;

 task body One is
    I: Integer := 1;
 begin
    while I <= 99 loop
       Put(I);
       Two.Go;
       I := I + 2;
    end loop;
 end One;
```

```
   task body Two is
      J: Integer := 2;
   begin
      while J <= 100 loop
         Put(J);
         accept Go;
         J := J + 2;
      end loop;
   end Two;

begin -- Hw
   null;
end Hw;
```