DATE 10/11/19 OM OT OW OT OF OS OS

PL HW1

1 NOTES

1. Provide REs for defining the syntax of the following. You can only use concatenation, |, *, parentheses, ε (the empty string), and expressions of the form [A-Z], [a-z0-9], etc. to create REs. You cannot use + or any kind of count variable.

(a) Strings consisting of any number of lower case letters, exactly two upper case letters, and exactly one digit, such that the digit is somewhere between the two upper case letters. They can be of any length. e.g. "doxFexo7wvsQdle" would be a valid string.

Solution: [a-z]* [A-Z] [a-z]* [0-9] [a-z]* [A-Z] [a-z]*

(b) Floating point literals that specify an exponent, such as 243.876E11. There must be at least one digit before the decimal point and one digit after the decimal point (before the "E").

Solution: ([1-9][0-9]* | 0). [0-9][0-9]* E [1-9][0-9]*

(c) Procedure names that: must start with a letter; may contain letters, digits, and _ (underscore); and must be no more than 7 characters.

Solution: [a-zA-Z] (ε | [a-zA-Z0-9_] (ε | [a-zA-Z0-9_] (ε | [a-zA-Z0-9_] (ε | [a-zA-Z0-9_] (ε | [a-zA-Z0-9_] (ε | [a-zA-Z0-9_])))))

where [a-zA-Z0-9_] := ([a-zA-Z0-9] | _)

2.(a) Provide a simple CFG for the language in which the following program is written. You can assume that the syntax of names and numbers are already defined using REs (i.e. You don't have to define the syntax for names and numbers).

```
x: int;
fun f (x: int, y: int)
    z: int;
{
    z = x+y-1;
    z = z+1;
    return z;
}
proc g()
    a: int;
{
    a = 3;
    x = f(a, 2);
}
```

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

Solution: ① Terminal Symbols : { int, fun, (,), :, ,, ;, {, }, =, +, -, return, proc }.

② Non-terminal Symbols : { PROG, VAR_DEF, FUN_DEF, FUN_TITLE, STMTS, ASSIGN, RET, PROC_DEF, STMT PROC_TITLE; CALL, NAME, NUMBER, BODY, , ARG_LIST, EXPR, OPERATOR }

③ Productions

<PROG> ::= <VAR_DEF> <FUN_DEF> <PROC_DEF>

<VAR_DEF> ::= NAME : int ;

<FUN_DEF> ::= <FUN_TITLE> <BODY>

<FUN_TITLE> ::= fun NAME (<ARG_LIST>)

<ARG_LIST> ::= ε | NAME : int , NAME : int

<BODY> ::= <VAR_DEF> { <STMTS> }

<STMTS> ::= <STMT> <STMTS> | <STMT>

<STMT> ::= ε | <ASSIGN> | <RET>

<ASSIGN> ::= NAME = <EXPR> ;

<EXPR> ::= NAME <OPERATOR> <EXPR>

| NAME | NUMBER | CALL

<OPERATOR> ::= + | -

<CALL> ::= NAME (NAME , NUMBER) ;

<RET> ::= return NAME ;

<PROC_DEF> ::= <PROC_TITLE> <BODY>

<PROC_TITLE> ::= Proc NAME (<ARG_LIST>)

(b) Draw the parse tree for the above program.

Solution: see picture attached.

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

Static scoping: Names used in a function are resolved in the environment of the function definition.

Dynamic scoping: Names used in a function are resolved in the environment of the function call.

(b) Give a simple example that would illustrate the difference between static and dynamic scoping.

Solution:

```
def f():
    x: int := 1
    def g():
        print(x)
    def h():
        x: int := 2
        g()
h()
```

Static scoping: print 1

Dynamic scoping: print 2

(c) In a block structured, statically scoped language, what is the rule for resolving variable references ?

Solution: The program traces each block from the innermost block to the outermost block for the function definition. In the example provided in (b), g and h are both defined within f. So when we call h which in-turn calls g, g searches in its block and finds no variable x. So g goes to its outer block for its definition. Which is f, and finds the declaration of variable x in f's block.

(d) In a block structured but dynamically scoped language, what is the rule for resolving variable references ?

Solution: The program traces down the call stack, if it doesn't find the variable declaration in its own stack frame, it searches in its caller's stack frame, and so on, until it reaches the bottom of the stack. Use (b) as an example, h calls g. g first searches its own stack frame but cannot find the declaration for variable x. so g goes to search in its caller, which is h's stack frame and finds the declaration for variable x in h's stack frame.

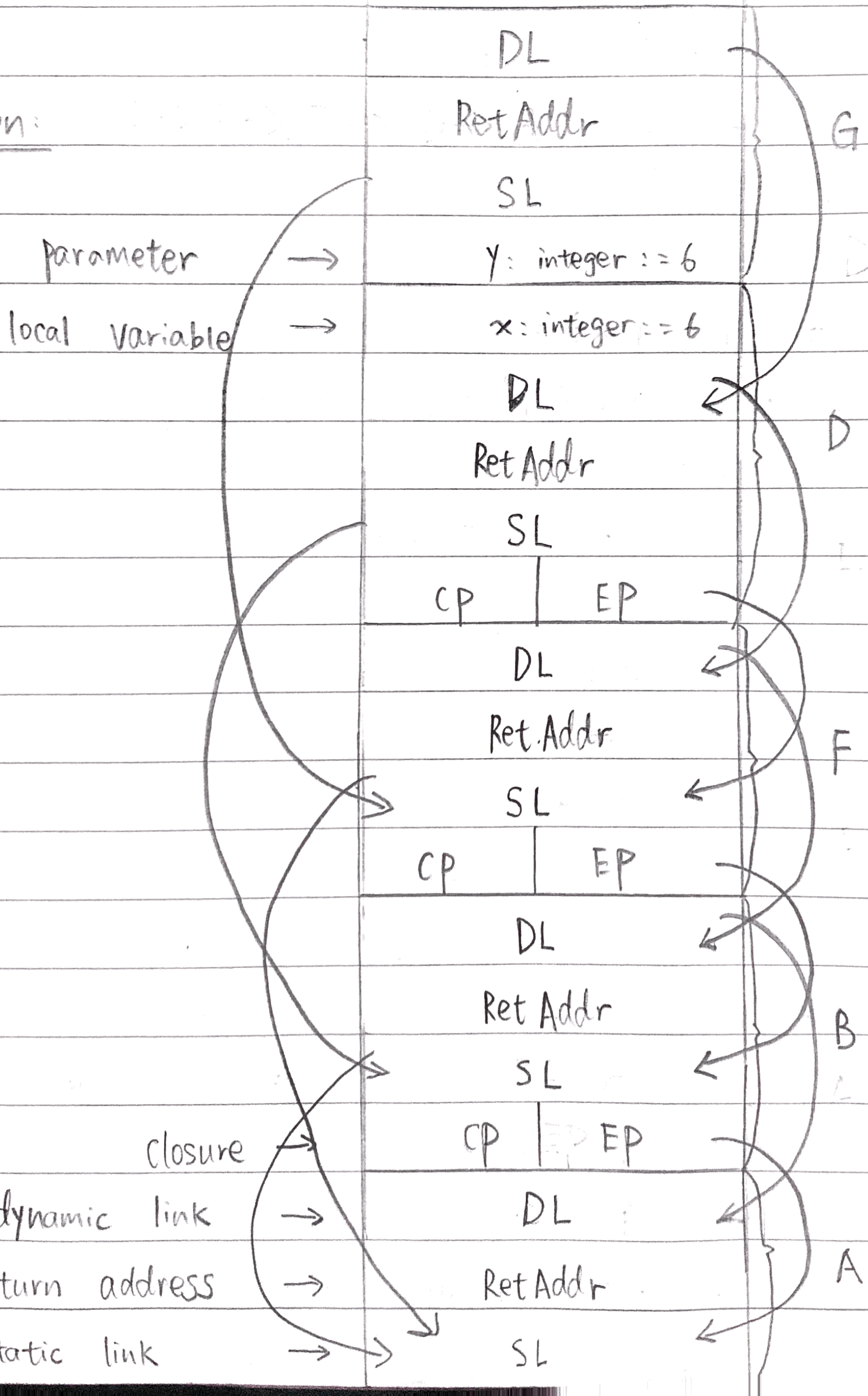
4. (a) Draw the state of the stack, including all relevant values (e.g., variables, return addresses, dynamic links, static links, closures), at the time that the writeln(y) is executed.

```
Procedure A;
  procedure B (procedure C)
    procedure D (procedure I);
      X: integer := 6
    begin (* D *)
      I(x);
    end;
  begin (* B *)
    (D);
  end;

  procedure F (procedure H);
    procedure G (y: integer)
    begin (* G *)
      writeln (y);
    end;
  begin (*F *)
    H(G);
  end;
```

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

Solution:



5. For each of these parameter passing mechanism,

(a) pass by value

(b) pass by reference

(c) pass by value-result

(d) pass by name

State what the following program would print.

```
program foo;

var i, j : integer;
    a    : array [1..5] of integer;

procedure f (x, y : integer)
begin
    x := x * 2;
    j := i + 1;
    y := a[i] + 1;
end

begin
    for j := 1 to 5 do a[j] = j * 2;
    i := 2;
    f(i, a[i]);
    for j := 1 to 5 do print (a[j]);
end
```

Solution:

(a) pass by value: 2, 4, 6, 8, 10

(b) pass by reference: 2, 11, 6, 8, 10

(c) pass by value-result: 2, 7, 6, 8, 10

(d) pass by name : 2, 4, 6, 8, 11

6. (a) See attached code.

(b) looking at the code for (a), are the printing of any of the numbers occurring concurrently? Justify your answer by describing what concurrency is and why these events do or do not occur concurrently.

Solution: Do not have concurrency.

Task One first prints its first 10 numbers, then it instructs Task Two to print its first 10 numbers, after which Task One prints the next 10 numbers again. And this happens until Task One and Task Two have both printed all their numbers. The whole printing process occurs strictly sequentially, so no concurrency may exist since we know concurrency means no assumptions can be made about the order of executions of two threads. If we had concurrency, we would have no guarantee that the two tasks print 10 numbers one by one.