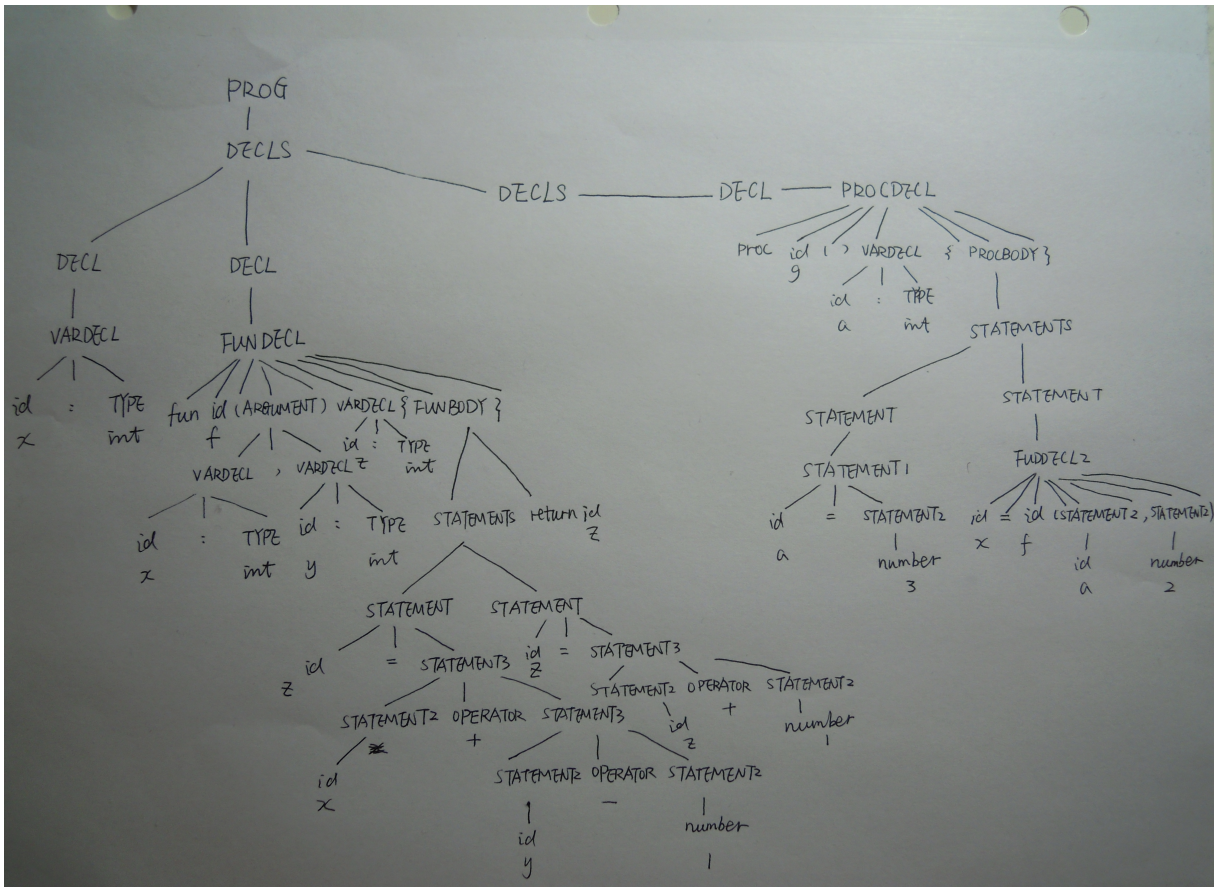
Figure 1: Parse tree for Problem 2



1: The First Problem

(a) $[a-z]^*$[A-Z]$[a-z]^*$[0-9]$[a-z]^*$[A-Z]$[a-z]^*$

(b) $[+-]^?[0-9]^+\backslash.[0-9]^+$[E]$[+-]^?[0-9]^+$

(c) [A-Za-z][A-Za-z0-9_]{0,14}

2: The Second problem

(a)
PROG → DECLS
DECLS → DECL DECLS | DECL
DECL → VARDECL | FUNDECL | PROCDECL
VARDECL → id : TYPE
TYPE → int | float
FUNDECL → fun id (ARGUMENTS) VARDECL { FUNBODY }
ARGUMENTS → VARDECL, VARDECL
FUNBODY → STATEMENTS return id

PROCDECL → proc id ( ) VARDECL { PROCBODY}
PROCEBODY → STATEMENTS
STATEMENTS → STATEMENT STATEMENTS | STATEMENT
STATEMENT→ STATEMENT1 | FUNDECL2
FUNDECL2 → id = id (STATEMENT2, STATEMENT2)
STATEMENT1 → id = STATEMENT2 | STATEMENT3
STATEMENT2 → id | number
STATEMENT3 → STATEMENT2 OPERATOR STATEMENT2 | STATEMENT3
OPERATOR → + | - | * | /

(b) The parse tree is shown on the first page.

3: The Third problem

(a) Static scoping: Variables and functions are determined at compile time. They are determined by the local block where they are defined in the source code.

Dynamic scoping: Variables and functions are determined at running time. They are determined by the caller (father block), not where is the block in the source code.

(b) The same example with different scoping:

```
program A( ) {
int x = 0;
void B( ){
     x = x + 2;
}
{
int x = 6;
B( );
}
print (x);
}
```

Static scoping: x= 2 is printed.
We assume int x = 0 is x1, int x= 6 is x2. Because B is defined in program A, so x2 will not influence the value of x in B. After calling B, x1 = x1 + 2 = 2. So print x1 = 2.

Dynamic scoping: x = 0 is printed.
Because print(x) is in scope A, x1 is the local variable in A, so x1 = 0 is printed. x2 and B are in the same scope, so the value of x in B is x2.

(c) With static scoping, variable is determined by the location of its definition in the source code. If variable is defined inside a block, the valid binding for this variable is the one in the local environment. If such valid binding doesn't exist in the local block, then we should look for valid binding from the block containing the starting block. If the binding is found in this block, it is the valid one. If not, we still keep looking for from the nearest block to the furthest block. As the example above, B is defined in A. So the valid binding for x in B is x1 =0.

(d) With dynamic scoping, variable is determined by the running time. So the variable is determined by the caller, the caller means father block. B and x2 are in the same block, so when

B is called , x2 is the valid binding for B.

4: The Fourth problem

(a) The state of stack is shown on the last page.

(b) In JavaScript, a closure can access to the outer function's variables even when outer function returns. Variables will be stored on heap even if outer function is executed. If closure is stored on the stack, the variables will be destroyed after outer function is executed.

```
program A( ) {
    var expression = " The date is ";
    void B( ){
        var date = "March 7th";
        print (expression + date);
    }
    return B;
}
var theDate = A( );
theDate;
```

As the example above, function B is returned in the outer function before execution. theDate becomes a closure to store the function and its environment on the heap.

5: The Fifth problem

(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: The Sixth problem

(a)

```
with text_io; use text_io;

procedure main is

package int_io is new integer_io(integer); use int_io;

task one is
    entry oneDo;
end one;

task two is
    entry TwoDo;
```

```
end two;

task body one is
begin
    accept OneDo do
        null;
    end;
    for I in 1 .. 1000 loop
        put(I);
        if I mod 100 = 0 then
            two.TwoDo;
            accept OneDo do
                null;
            end;
        end if;
    end loop;
end one;

task body two is
begin
    accept TwoDo do
        null;
    end;
    for J in 2001 .. 3000 loop
        put(J);
        if J mod 100 = 0 then
            one.OneDo;
            accept TwoDo;
                null;
            end;
        end if;
    end loop;
end two;

begin
one.OneDo;
end main;
```

(b) The printing of the numbers are not occurring concurrently in the code above. Because I have already defined the printing order. There are two entries both in task one and task two. When the main procedure calls the task one's entry, task one starts to print until it satisfies the if statement. And then it calls task two's entry, so task two starts to print and task one pauses. Until task two satisfies the if statement, task two calls task one's entry and task two pauses. Then task one continues print until it satisfies the if statement. So task one and task two are not running concurrently in this case.

Figure 2: The state of stack for Problem 4