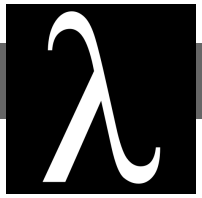


Out: Tuesday, 30 May 2023 at 18:00 ET
Due: Thursday, 8 June 2023 at 23:59 ET

Total Points: 102

Contents

1 Datatypes and Types

For each of the following SML declarations, what is the type of the value being declared? If one of the expressions is not well-typed, briefly explain why.

Note: Though you can just plug the expression into the SML/NJ REPL and have it do all the work for you, remember that we might ask you to solve this kind of problem on an exam. We advise you to learn how to do it by hand, because you won't have a REPL on the exam.

Use this datatype to answer the following type questions.

```
datatype digit = Zero | One | Two | Three | Four
               | Five | Six | Seven | Eight | Nine
```

Task 1.1. (1 point)

```
1 + One
```

Task 1.2. (1 point)

```
Five + Two
```

Task 1.3. (1 point)

```
fn 0 => Zero
```

Use the following datatypes to answer the next few type questions.

```
datatype professor = Steve | Mike
datatype job = Student | Professor of professor | TA of string
```

Task 1.4. (1 point)

```
Student
```

Task 1.5. (1 point)

```
TA
```

Task 1.6. (1 point)

```
fun promote (x : job) =
  (case x of
    Student => TA "Polly"
  | TA y => Professor y
  | z => z)
```

Use the following datatype to answer the next few type questions.

```
datatype tritree = Empty | Node of int * tritree * tritree * tritree
```

The `int option` type is comprised of two constructors: `NONE` and `SOME x`, where we have `x : int`.

Task 1.7. (1 point)

```
fun numBranches (Node(x, L, M, R)) =
  1 + numBranches L + numBranches M + numBranches R
```

Task 1.8. (1 point)

```
fun max (Empty : tritree) = NONE
  | max (Node (x, L, M, R)) =
    let
      val currmax = Int.max (Int.max (max L, max M), max R)
    in
      SOME (Int.max (x, currmax))
    end
```

Note that `Int.max : int * int -> int`.

2 Total `fun`

Recall the following definitions:

Definition. An expression `e` is *valuable* iff there exists a value `v` such that `e` $\Longrightarrow$ `v`.

Definition. A function `f : t1 -> t2` is *total* iff for all values `x : t1`, `f x` is valuable.

A thorough understanding of these concepts is ~~imperative~~[1] crucial for this task and future problems. In particular, you are expected to correctly cite totality and valuability in your proofs to reflect SML's order of evaluation.

SML is *eagerly evaluated*, which means that function arguments are evaluated before passing them into the function body. Consequently, if your proof steps through the body of a function without first evaluating its argument(s), you *must* prove that its argument(s) are indeed valuable. Sometimes this requires showing that an associated function is total.

Refer to this Totality Help Sheet for additional help on how to cite totality in proofs.

2.1 Totality and valuability

Let `t1`, `t2`, `t3` be non-function types (e.g. `string` or `int list`, but not `int -> int` or `int list -> int`).

Decide whether each of the following statements are true or false. If a statement is true, briefly justify why in a sentence or two. If you say it is false, provide a counterexample. No verbose explanation is needed.

Let's use the following statement as an example:

There is no function `f` such that `f x` $\Longrightarrow$ `0` for all values `x : int`.

The following solution would receive full points:

False. `fun f _ = 0`.

Task 2.1. (1 point)

Let `f : t1 -> t2` be a value. If there exists a value `x : t1` such that `f x` is valuable, then `f` is total.

Task 2.2. (1 point)

Let `f : t1 -> t2` be a total function. Then `f` is valuable.

Task 2.3. (1 point)

Let `x : t1` be a value. Then for all `f : t1 -> t2`, `f x` is valuable.

Task 2.4. (2 points)

Let `f : t1 -> t2` be a value. If for all values `y : t2`, there exists an `x : t1` such that `f x` $\cong$ `y`, then `f` is total.

[1] We don't do that here.

Task 2.5. (2 points)

Let `f : t1 -> (t2 -> t3)` be total. Then for all values `x : t1`, `f x` is valuable.

Task 2.6. (2 points)

Let `f : t1 -> (t2 -> t3)` be total. Then there exists a value `x : t1` such that `f x` is total.

3 Sum Nights

3.1 Sublists

We'll begin by defining what it means for a list `L1` to be a *sublist* of `L2`, denoted `L1` $\sqsubseteq$ `L2`.

Intuitively, we know what this means: `[1,3,5,6]` is a sublist of `[1,2,3,4,5,6]` because every element of `[1,3,5,6]` is also in `[1,2,3,4,5,6]`, and they appear in the same order.

There are possibly many different ways we could formalize this (you're encouraged to think about it), but here's how we decided to do it (inductively, as usual):

Definition. For all types `t`, $\sqsubseteq$ is defined as

1. For all `L : t list`, `[]` $\sqsubseteq$ `L`
2. For all `A : t list`, `L : t list` and all `x : t`, if `A` $\sqsubseteq$ `L`, then `x::A` $\sqsubseteq$ `x::L`
3. For all `A : t list`, `L : t list` and all `x : t`, if `A` $\sqsubseteq$ `L`, then `A` $\sqsubseteq$ `x::L`

If `L1 : t list, L2 : t list` such that `L1` $\sqsubseteq$ `L2`, then we say that `L1` is *a sublist of* `L2`.

So, we know that `[1,3,5,6]` is a sublist of `[1,2,3,4,5,6]` because `[3,5,6]` is a sublist of `[2,3,4,5,6]`, and we know that `[3,5,6]` is a sublist of `[2,3,4,5,6]` because `[3,5,6]` is a sublist of `[3,4,5,6]`, and so on.

3.2 Sublist Sum

Given `L : int list`, we define the sum of `L` with the following SML code:

```
fun sum [] = 0
  | sum (x::xs) = x + sum xs
```

so, for example, `sum [1,2,3]` $\Longrightarrow$ `6`.

There is a famous problem in computer science known as the **sublist sum** problem: given a list of integers `L` and an integer `n`, identify if there exists a sublist `L'` of `L` such that `sum L'` $\cong$ `n`.

Let's solve the sublist sum problem in SML! You will not only determine if a sublist exists which sums to the desired value, but you will also determine what that sublist is if it exists! We can do this using the `option` datatype introduced in lab.

Task 3.1. (16 points)

In `code/sublist-sum/sublistSum.sml`, write the function

```
sublistSum : int list * int -> int list option
```

REQUIRES: true

ENSURES: `sublistSum (L, n)` $\Longrightarrow$ `SOME L'` where `L'` $\sqsubseteq$ `L` and `sum L'` $\cong$ `n`.
`sublistSum(L, n)` $\Longrightarrow$ `NONE` if there is no such sublist `L'`.

Before you begin, here are some things to keep in mind:

- As a convention, the empty list `[]` has a sum of 0.
- *Hint*: each element of the input list is either in the sublist, or it isn't.
- In some instances of the problem, you might find that there are multiple correct answers. For example, `sublistSum([~1, 1], 0)` could reasonably return `SOME [~1,1]` or `SOME []`. We will consider both correct.
- Don't forget the ordering requirement for sublists: `[2,1]` is not a valid sublist of `[1,2,3,4,5,6]` as the `2` does not come after the `1` in the original list.
- It's easy to produce correct and unnecessarily complicated functions to compute sublist sums. It's almost certain that your solution will have $O(2^n)$ work, so don't try to optimize your code too much. There is a very clean way to write this in a few elegant lines.

4 #treedeep5me

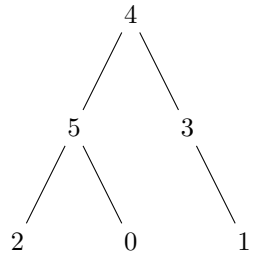
We define a subtree inductively as follows:

Definition (Subtree)**.** Let `T : tree` be a value.

1. `T` is a subtree of `T`.
2. Let `T' : tree` be a value. If `T'` is a subtree of `T` and `T' ≅ Node (L,x,R)`, then `L` and `R` are subtrees of `T`.

As usual, convince yourself that this definition of subtrees is equivalent to the intuitive one.

The ***lowest common ancestor*** (lca) between two nodes in a tree is the tree defined by the root of the smallest subtree that contains both nodes. To further your understanding, consider the following examples:



- 4 is the lowest common ancestor of 2 and 3
- 4 is the lowest common ancestor of 1 and 5
- 4 is the lowest common ancestor of 1 and 2
- 5 is the lowest common ancestor of 0 and 5
- 5 is the lowest common ancestor of 0 and 2
- 3 is the lowest common ancestor of 1 and 3

In this section, you will be given a tree that does not contain any duplicate values, and two target values that may or may not be in the tree. Our goal is to implement a function to solve the lca problem.

It seems costly to repeatedly check whether a value exists in our tree. Our approach will instead use the path through the tree to each of our target nodes. To achieve this, we introduce the following datatype:

```
datatype direction = LEFT | RIGHT
```

where `LEFT` indicates traversal through a node's left subtree, and `RIGHT` indicates traversal through a node's right subtree.

4.1 Testing your code

When writing tree test cases, it (wood) be annoying to con(cedar) where all the nested parentheses go. So, to write test cases, feel free to use this Tree-to-SML converter:

https://leesue630.github.io/tree-to-sml-converter/

Input the depth, (cy)press 'Generate SML Text', and you will(ow) be able to copy-paste the SML tree into your code.

If you want to test your code using the testing library, you can use `Test.general_eq`. **This function works for any type you need for this assignment!** For example, if you have a function `f: int -> tree`, and you wanted to check that evaluating `f 0` outputs a tree that matches some value `expectedT : tree`, you would do

```
val () = Test.general_eq("Tree Test", expectedT, f 0)
```

If your test case failed and you wanted to check out what `f 0` looks like, you can remove the test case, run `./smlnj sources.mlb`, and type in `f 0` in the REPL. Then, go to the Tree-to-SML converter linked above, go to the SML tab, and copy-paste your output there.

4.2 Additional hints

For the following tasks, it's important to make sure that your functions are ***maximally parallel*** so that they have the best possible asymptotic work/span. The primary thing to keep in mind is that we consider tuples as being evaluated in parallel.

Consider this (somewhat contrived) example:

```
fun someExpensiveFn (x : int) : int = (* some expensive function *)

val sequential : int =
    let
      val x = someExpensiveFn 1 (* SML fully evaluates this... *)
      val y = someExpensiveFn 2 (* ...then evaluates this. *)
    in
      x + y
    end

val parallel : int =
    let
      (* SML evaluates both of these simultaneously *)
      val (x,y) = (someExpensiveFn 1, someExpensiveFn 2)
    in
      x + y
    end

(* we pass a tuple to the + function, so doing this is also
   maximally parallel *)
val parallel2 : int = (someExpensiveFn 1) + (someExpensiveFn 2)
```

Be mindful of your order of evaluation when writing your code for the following tasks, and be sure to optimize things to be more parallel if possible.

For each of the following tasks:

- Your code should be maximally parallel.
- Your code shouldn't need to define any additional helper functions besides the ones you will implement.

Task 4.1. (2 points)

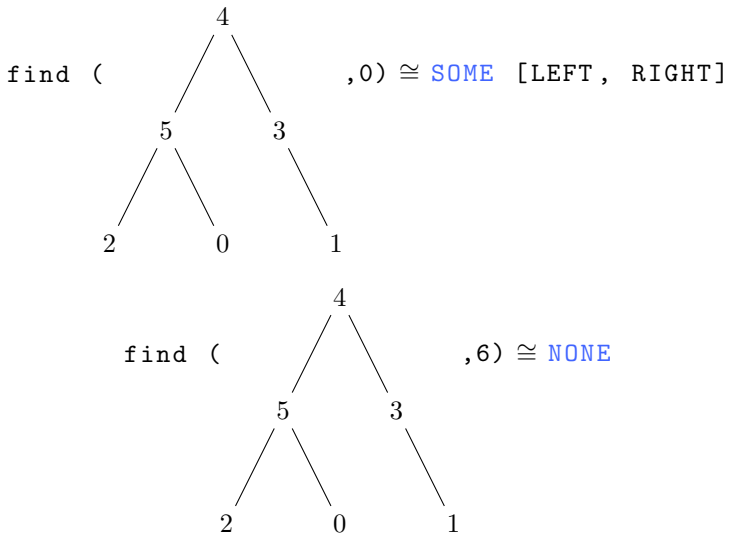
In `code/lca/lca.sml`, write the function

```
find : tree * int -> direction list option
```

REQUIRES: `T` contains no duplicates

ENSURES: `find (T, v)` ≅ `NONE` if `v` is not in `T`, else `SOME L`, where `L` is a list of directions that can be used to traverse from the root of `T` to `v`

To clarify the behavior of `find`, consider the following example:



Say that we find a path to one of our target nodes in the tree and produce the direction list `L`. If we traverse our input tree according to the directions in any prefix of the list `L`, we arrive at an ancestor of our target. Since our goal is to find the lowest common ancestor of our two target nodes, we want to use the longest prefix that is common to both paths. To achieve this, we will use two more helper functions.

Task 4.2. (2 points)

In `code/lca/lca.sml`, write the function

```
follow : tree * direction list -> tree option
```

REQUIRES: true

ENSURES: `follow (T, L)` ≅ `NONE` if traversing T according to the directions in `L` do not lead to a valid subtree of `T`, else `SOME T'`, where `T'` is the subtree of `T` that is obtained by

traversing `T` according to `L`

Task 4.3. (2 points)

In `code/lca/lca.sml`, write the function

```
common : direction list * direction list -> direction list
```

REQUIRES: true

ENSURES: `common (L1, L2)` $\cong$ `L`, where `L` is the longest prefix that is common to both `L1` and `L2`

Constraint: Your implementation of `common` must have $O(\min(|\texttt{L1}|, |\texttt{L2}|))$ work and span.

Task 4.4. (6 points)

Finally, use `find`, `follow`, and `common` to implement the function

```
lca : tree * int * int -> tree option
```

REQUIRES: `T` contains no duplicates, `a <> b`

ENSURES: `lca (T,a,b)` $\cong$ `NONE` if `a` or `b` are not in `T`, else `SOME (Node (L,x,R))` s.t. `Node (L,x,R)` is a subtree of `T`, and either:

- `a` is in `L` and `b` is in `R`
- `b` is in `L` and `a` is in `R`
- `a` = `x` and `b` is in `L` or `R`
- `b` = `x` and `a` is in `L` or `R`

Task 4.5. (8 points)

Write and solve recurrences for the work and span of your implementation of `find`, `follow`, and `lca` in terms of the number of nodes n in the input tree. For this analysis, assume that the input tree is balanced. That is, for all subtrees of the form `Node (L,x,R)`, the number of nodes in `L` and `R` differ by at most 1. Please include a copy of your implementation in your written submission, and show your work when solving for the big-O bounds.

You may assume that `common (L1, L2)` has $O(\min(|\texttt{L1}|, |\texttt{L2}|))$ work and span. **Note: when indicating the work of common, you should still use $W_{common}(k_1, k_2)$ where k_1, k_2 are in terms of n because we are working with the number of nodes, n.**

Constraint: You must use the **tree method** to solve your recurrences.

5 The Flat List Society

The recently-founded Flat List Society has been closely following your progress in 15-150. In the previous homework, they (like you) saw the `int list` type, and saw proofs of functions that take `int list`s as input and produce `int list`s as output. However, they have recently learned that...

The same techniques that we have been using to program with `int list`s can be used for `list`s containing other types of elements - in particular, the elements themselves can be `int list`s! Values like this have the type `(int list) list`, which can also be written `int list list`, representing lists of lists of integers. For instance:

```
          [1]::[] ≅ [[1]]                    : int list list
[1,2]::[[3],[4,5]] ≅ [[1,2],[3],[4,5]]      : int list list
```

After having learned about this, the self-respecting members of the Flat List Society have become extremely offended - they only like 'flat' `list`s, that is, `list`s that do not have `list`s as elements. They have since threatened to use a `list` of weapons on the course! Help us appease them by writing a function that can flatten our `list`s!

5.1 We live in a (flat) society

Before we begin this heroic task, we will need to consider what it means for a list to be "flattened". The Flat List Society points you in the direction of an ancient tome that contains a function known to flatten lists.

Definition. A list `F : int list` is flattened with respect to another list `LL : int list list` if `oldFlat LL = F` where `oldFlat` is defined as follows:

```
fun oldFlat [] = []
  | oldFlat (L::LS) = L @ (oldFlat LS)
```

Task 5.1. (8 points)

Unfortunately, in today's anti-`@` world, this function has been deemed illegal. Armed with this ancient function as our definition, we must provide a new function allowing the Flat List Society to flatten their `int list list`s!

In `code/flatten/flatten.sml`, write the function

`flatten : int list list -> int list`

REQUIRES: true

ENSURES: `flatten LL` $\Longrightarrow$ `F` such that such that F is flattened with respect to LL

Constraint: You may not use or create any helper functions for this task. This includes `@`, `length` and any functions from the standard library.

Here are some example outputs:

```
flatten [] = []
flatten [[]] = []
flatten [[15, 150]] = [15, 150]
flatten [[15150]] = [15150]
flatten [[], [15, 15], [], [], [], [0]] = [15, 15, 0]
flatten [[1, 5], [1], [], [5, 0]] = [1, 5, 1, 5, 0]
```

5.2 We can actually prove it's flat (unlike some other societies)!

However, it is not enough to have written `flatten`. The Flat List Society has aggressively demanded that you prove to them that your function actually produces flattened lists! They will only accept your function if it behaves the same as their sacred function from an era long past. Help us fend off the impending attack!

Task 5.2. (15 points)

Using your implementation, prove the following:

Theorem. For all values `LL : int list list`, `flatten LL` $\cong$ `oldFlat LL`.

You may assume that `@` and `oldFlat` have the following implementations:

```
fun []      @ L = L
  | (x::xs) @ L = x::(xs @ L)

fun oldFlat [] = []
  | oldFlat (L::LS) = L @ (oldFlat LS)
```

Lemma 5.1. You may cite that `oldFlat` is total.

Hint: Think about the structure of the code when structuring your proof. Your proof should mirror the code.

6 Compiler Optimizations

In this problem, we will be working with the following datatype, which represents an arithmetic expression over integers.

```
datatype exp
  = Var of string
  | Int of int
  | Add of exp * exp
  | Mul of exp * exp
  | Not of exp
  | IfThenElse of exp * exp * exp
```

Here is the meaning of each constructor in the datatype:

Constructor	Meaning
`Var x`	A variable with name `x`. When evaluating the expression, this variable must be assigned a value in the environment.
`Int n`	A constant integer with value `n`.
`Add (a,b)`	The sum of two expressions, `a` and `b`.
`Mul (a,b)`	The product of two expressions, `a` and `b`.
`Not a`	The "truthy" negation of an integer expression. If `a` is `0`, `Not a` should be `1`. Otherwise, `Not a` should be `0`.
`IfThenElse (i,t,e)`	"Truthy" casing on integers. If `i` is nonzero, the expression evaluates to `t`. Otherwise, it evaluates to `e`.

Here are some example representations of operations.

Arithmetic	`exp`
$2+3$	`Add (Int 2,Int 3)`
$2+x$	`Add (Int 2,Var "x")`
$3x+1$	`Add (Mul (Int 3,Var "x"),Int 1)`
$3 \cdot 2+x$	`Add (Mul (Int 3,Int 2),Var "x")`
"if $x \neq 0$ then y else 8"	`IfThenElse (Var "x",Var "y",Int 8)`
"1 if $x=0$, else 0"	`Not (Var "x")`
"0 if $x=0$, else 1"	`Not (Not (Var "x"))`

Environments We also need a way to represent the "environment", which contains the variable bindings. We will use the `environ` type alias:

```
type environ = (string * int) list
```

Note that there are two helper functions defined in `code/expressions/exp.sml` that you might find useful as you work through the following tasks.

- `lookup (env,x)` finds the integer bound to variable `x` in environment `env`.
- `vars e` evaluates to a list of all variable names in expression `e`.

Full specifications and implementations can be found in the file itself.

6.1 Evaluation

Before we do anything, it would be helpful to define an evaluation function for our datatype. Our function will take in an environment, which is just a list of `string * int` pairs representing which integer each variable name is bound to. You may assume the list has no duplicates.

Task 6.1. (5 points)

In `code/expressions/exp.sml`, write:

> `eval : environ * exp -> int`
>
> REQUIRES: Each variable in `e` has one entry in `env`
>
> ENSURES: `eval (env,e)` $\Longrightarrow$ `n`, where `n` is the integer representing the value of `e` given environment `env`

For example, it should be the case that:

```
eval ([("x",45),("y",3)],Add (Int 15,Mul (Var "x",Var "y"))) ⟹ 150
```

If you attempt to view deeply-nested `exp` values, the REPL will stop printing a few layers in, instead showing a `#` sign. You can (temporarily) increase the print depth by typing the following into the REPL:

```
Control.Print.printDepth := 100
```

or by running in the terminal:

```
PRINT_DEPTH=100 ./smlnj exp.sml
```

6.2 Constant Fusion

Many programming language implementations have optimizations so that the code you write will run faster. For example, if you write the Python program

```
def foo(x):
    return x + (2 * 3)
```

an optimization called *constant fusion* may be applied, transforming the function to

```
def foo(x):
    return x + 6
```

In this case, the constant-fused expression has precomputed `(2 * 3)`, so we can avoid the multiplication step at runtime when the function is called. We've implemented a similar optimization[2] in `code/expressions/fuse.sml` that fuses multiplications for our integer expression language:

```
fun fuse (Var x) = Var x
  | fuse (Int n) = Int n
  | fuse (Add (a,b)) = Add (fuse a, fuse b)
  | fuse (Mul (a,b)) = (
      case (vars a,vars b) of
        ([],[]) => Int (eval ([],a) * eval ([],b))
      | _       => Mul (fuse a,fuse b)
    )
  | fuse (Not a) = Not (fuse a)
  | fuse (IfThenElse (i,t,e)) = IfThenElse (fuse i,fuse t,fuse e)

val () = Test.general_eq("Fuse Test",
                         Add (Var "x",Int 6),
                         fuse (Add (Var "x", Mul (Int 2,Int 3))))
```

However, before enabling this optimization, we want to ensure that the constant-fused code is actually equivalent to the original code. Thankfully, we can prove this fact and sleep soundly for the rest of our lives.

The next four tasks will refer to the validity of `fuse`.

Theorem (Validity of `fuse`)**.** For all values `env` and `e` satisfying the REQUIRES for `eval`,

$$\texttt{eval (env,fuse e)} \cong \texttt{eval (env,e)}$$

.

You may make use of the following lemmas:

Lemma 6.1 (Totality of `vars`)**.** The `vars` function is total.

Lemma 6.2 (Totality of Fusion)**.** The `fuse` function is total.

Lemma 6.3 (Idempotence of Fusion)**.** For all `e : exp`:

$$\texttt{fuse e} \cong \texttt{fuse (fuse e)}$$

Lemma 6.4 (Closed Expression Valuability)**.** For all `e : exp`, if `vars e` $\cong$ `[]`, then for all `env : environ`, we have that `eval (env,e)` is valuable.

Lemma 6.5 (Environment Independence of Closed Expressions)**.** For all `e : exp`, if `vars e` $\cong$ `[]`, then for all `env : environ`, we have that `eval (env,e)` $\cong$ `eval ([],e)`.[3]

[2] This is just one of many possible optimizations! For example, you could consider a similar optimization for addition, or optimizing expressions like `Add (Add (x,x),x)` to `Mul (Int 3,x)`. We will only consider the given optimization for constant multiplication on multiplication in this problem, but feel free to experiment with coding (and proving) other optimizations.

[3] Note that this is a special case of a stronger lemma. Namely, given `e : exp`, if `env` and `env'` bind all variables in `e` to the same integers and satisfy the REQUIRES of `eval`, then `eval (env,e)` $\cong$ `eval (env',e)`.

Feel free to abbreviate `IfThenElse` as `ITE`.

Note that this proof may depend on your implementation of `eval`. Make sure you are convinced of its correctness!

Task 6.2. (1 point)

Identify which cases are base cases for the structural induction proof of the theorem.

Task 6.3. (4 points)

State the inductive cases and their IHs for the structural induction proof of the theorem.

Task 6.4. (5 points)

Prove the `IfThenElse` case for the structural induction proof of the theorem.

Task 6.5. (11 points)

Prove the `Mul` case for the structural induction proof of the theorem.

Constraint:

- While you may use abbreviations to shorten your proof, make sure this is ***not*** at the expense of correctness clarity. In particular, please make the structure of your proof is clear: be sure to ***explicitly*** state all cases and inductive hypotheses.
- You are ***not*** allowed to cite the theorem itself in any step of the proof. (Note: this is slightly different from constructing and citing the Induction Hypothesis, which ***is*** allowed).

This assignment has a total of 102 points.