1 Language and Requirements

Sunbeam starts with the same semantics as Egg-Eater, minus closures, and adds support for a new type of heap-allocated objects: strings. The language features are:

- chars: storing chars as ascii values along with tag bits
- string expressions: creating strings, accessing chars, and mutating these chars
- printing strings and chars will print without any quotes, any quotes around string/char return values in this spec are purely for readability
- slice/range indexing operator: create a copy of the string or array from the specified indexes

2 Syntax Additions and Semantics

2.1 Concrete Syntax

Sunbeam adds two new value types: strings, which are heap allocated similarly to arrays, and chars which are static variables. We also are implementing range based indexing for both strings and arrays.

Additions to the concrete syntax:

```
‹expr›: ...

      | ‹string›

      | CHAR

‹char›: ...

      | IDENTIFIER = ‘‹expr›’

      | ‹expr› + ‹expr›

      | ischar ( ‹expr› )

‹string›: ...

      | IDENTIFIER = “‹expr›”

      | length ( ‹expr› )

      | isstring ( ‹expr› )
      | ‹expr› + ‹expr›
      | ‹expr› [ ‹expr› ]
      | ‹expr› [ ‹expr› ] := ‹expr›
      | ‹expr› [‹expr›..‹expr›]
‹array›: ...
      | ‹expr› [‹expr›..‹expr›]
```

2.2 Abstract Syntax

```
pub enum Prim {
      ...
      //two arguments
      Range
      //0 or more arguments
      MakeString
      //2 arguments
      HeapSet
      //1 argument
      HeapGet
      //1 argument
      IsChar
      //1 argument
      IsString
}
pub enum Exp {
      ...
```

```
    Char(char, Ann)
}
```

2.3 Transformations

No new transformations are required, all error checking is dynamic, and overloading the plus and Length operators is handled during compile_with_env.

3 Semantics and Representation of Strings

3.1 String Heap Layout

Strings are implemented on the heap largely the same as arrays were in diamondback, with the notable exception that they can only contain char values. Otherwise they are the exact same with the first element on the heap being the non-tagged length of the string and all subsequent elements being the chars within a string. Strings will not have a null terminator.

3.2 Tagging Chars and Strings

The string and char values will be tagged as such:

String: 0xWWWWWWWW[b011]
Char: 0xWWWWWWWW[b101]

NOTE: Chars are saved to the stack as their ASCII values then left shifted 3 and tagged with the above tag.

Example: ‘a’ has an ascii value of 97, so on the stack it will be 0b$\underline{01100001}$101, with the underlined bits being the original ascii value and the following bits being the tag.

3.3 String Equivalency

Just like with arrays, strings will be checked for pointer equality for their addresses on the heap.

4 Examples

String initialization:

```
let myString = “hello world!”
```

IMPORTANT NOTE:

Below example is *not* valid string initialization, this would instead be an array of chars

```
let strArr = [‘h’,’e’,’l’,’l’,’o’]
```

Char initialization:

```
let myChar = ‘a’
```

Mutating a string:

```
myString[0] := ‘a’
```

IMPORTANT NOTE:

Despite supporting ranged based indexing, it is not possible to overwrite a substring within a string with another string, you can only index into a string at one index and change it to a new char value

Range based indexing for strings and arrays:

```
let name = “Andrew Miller” in

name[0..5]
```

Would return “Andrew”

```
let nums = [1, 2, 3, 4, 5, 6] in

nums[3..5]
```

Would return [4, 5, 6]

Concatenation:

```
let x = “shoe” in x + ‘s’
```

Would return “shoes”

```
‘y’ + ‘o’
```

Would return “yo”

```
let name = “Milo Baumhardt” in


name[0..3] + “ R. “ + name[6..(Length(name) - 1)]
```

Would return “Milo R. Baumhardt”

5 Errors

The following will produce runtime errors:

- Using the -,*,add1 and sub1 prims on either a string or a char will raise an error: “arithmetic expected a number”
- <, <=, >, >= will raise an error: ”comparison expected a number”
- + on a string/char, nonstring/nonchar combo will raise an error: “arithmetic expected a number”
- Using &&,|| on a string or char will raise an error: “logic expected a boolean”
- Attempting to mutate a char in a string to a nonchar will raise the error: “attempted to set string index as nonchar”
- Attempting to go from a larger to a smaller index in range based indexing will raise the error: “range based indexing must be non-decreasing”
- Attempting to access a string/array at an invalid index, whether through standard or range based indexing will raise the error: “index out of bounds”
- Attempting to use the index/range index operator on non-string/array will raise the error: “indexed into non-string/non-array”
- Attempting to use the length prim on non-string/non-array will raise the error: “length called on non-string/non-array”
- Attempting to pass in non-num parameters to HeapGet or Range will raise the error: “index not a number”