README

Concrete Syntax

```
<prog> := <defn>* <expr>
<defn> :=
  | (fun <name> ((<name> <type>)*) <type> <expr>)
  | (struct <name> ((<name> <type>)+) )
<expr> :=
  | <number>
  | true
  | false
  | input
  | <identifier>
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (set! <name> <expr>)
  | (if <expr> <expr> <expr>)
  | (block <expr>+)
  | (repeat-until <expr> <expr>)
  | (<name> <expr>*)
  | (null <name>)
  | (alloc <name>)
  | (lookup <expr> <name>)
  | (update <expr> <name> <expr>)

<op1> := add1 | sub1 | print
<op2> := + | - | * | < | > | >= | <= | =

<type> := int | bool | <name>

<binding> := (<identifier> <expr>)
```

New syntax

- `type` can now include `<name>`, which indicates a struct.
- These structs are declared via the `<defn>` type, which is expanded to include a new keyword: `struct`.
  - Each `struct` has a name and **at least one** field, each field containing a name and a type.
  - Field names must be unique within a struct.
  - Field types can include any type, including `int`, `bool`, mutually-recursive structs, or any other struct.
  - Structs can be declared in any order; all struct names are parsed, then type-checking occurs.
  - Structs cannot share a name with a function

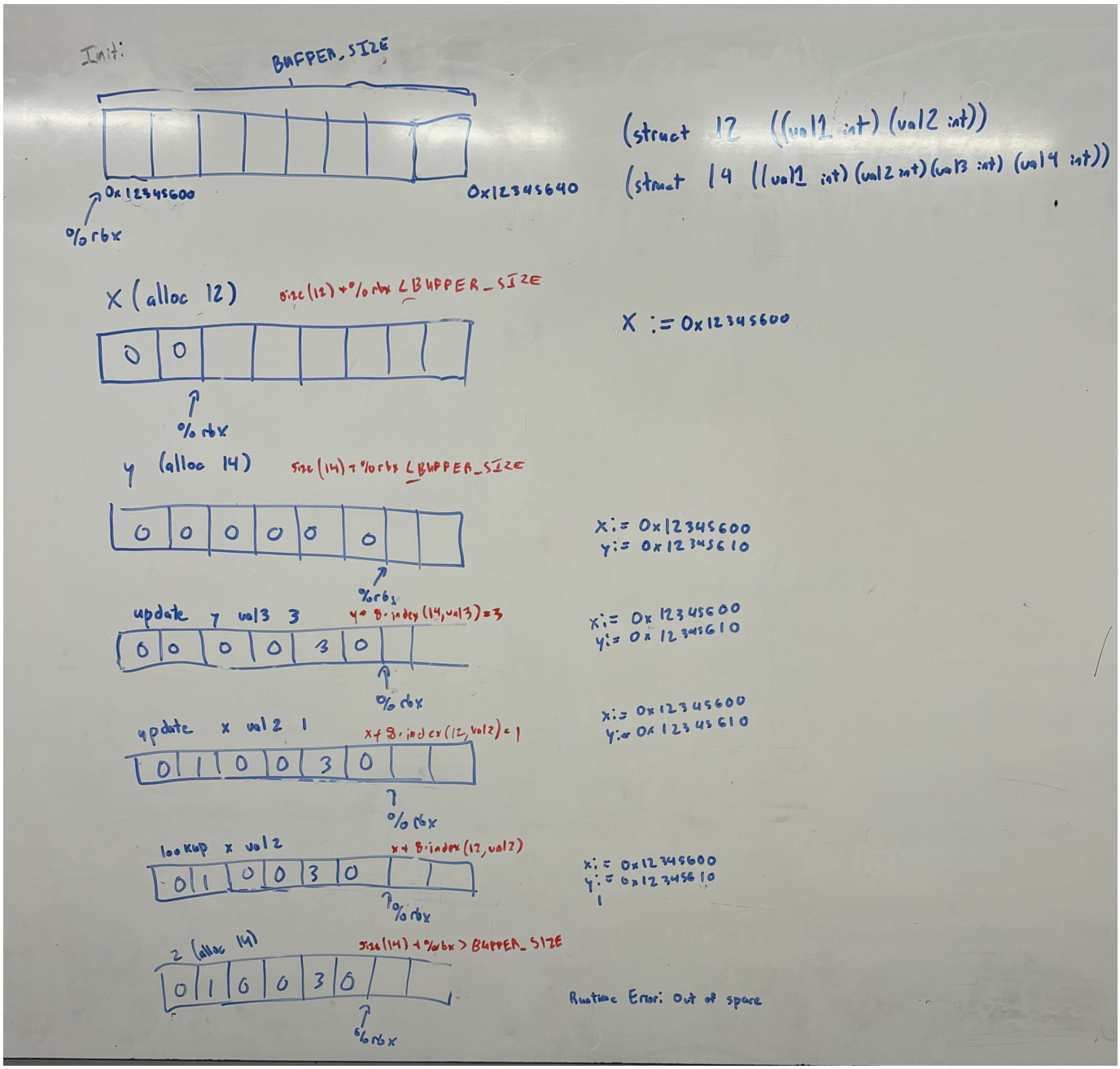
- Instances of structs, accessible via pointer, are created with `alloc`. Fields of new structs are initialized to 0 for integers, `null` for structs, and `false` for bools.
- Fields of structs can be updated and evaluated with `lookup` and `update`, both of which evaluate to the value in the struct field (the new value for `update`).
- `(null <name>)` creates a pointer of type `<name>` which has value `null`.
- `e1 = e2` for structs checks for pointer equality.
- The `print` keyword, when used on a struct type, points the following format:
  - Non-null pointer:

```
struct <name>
        <field_1_name> : <field_1_type> = <runtime_value>
        ...
        <field_n_name> : <field_n_type> = <runtime_value>
```

  - Null pointer:

```
null pointer to struct <name>
```

Diagram of Heap-Allocation



- We start with an empty array in the .bss section. %rbx is initialized as 0, indicating the beginning of the heap. The buffer has size 8 * 8 = 64. There are 2 possible structures: l2, which contains val1 and val2, and l4, which contains val1, val2, val3 and val4.
- Next, we allocate x, an l2 structure. There is still space in the .bss array (as %rbx + 8 * 2 $\leq$ `BUFFER_SIZE`), so the allocation works. All of x's fields are initialized as 0, while x is set to the address of the start of the .bss array. We then increment %rbx
- We then allocate y, an l4 structure. There is still space in the .bss array (as %rbx + 8 * 4 $\leq$ `BUFFER_SIZE`), so the allocation works. All of y's fields are initialized as 0, while y is set to the address of third entry of the .bss array. We then increment %rbx.
- Next, we update y's val3 field to 3. val3 is the 3rd field of y, so we update its effective address (y + 8 * 3) to 3.
- We then update x's val2 field to 1. val2 is the 2nd field of x, so we update its effective address (x + 8 * 2) to 1.
- Next, we lookup x's val2 field. It's effective address is (x + 8 * 2), and we load the value stored there into %rax.

- Finally, we try to allocate z, an l4 structure. There is no longer space in the .bss array (as %rbx + 8 * 2 > `BUFFER_SIZE`), so the allocation fails. We then throw the out of memory runtime error and the program terminates.

Required Tests

`input/simple_examples.snek`

```
% ./tests/input/simple_examples.run
1
2
2
```

`input/points.snek`

```
% ./tests/input/points.run
struct point (4297069072)
        x: int = 4
        y: int = 16
struct point (4297069088)
        x: int = 6
        y: int = 18
struct point (4297069104)
        x: int = 8
        y: int = 20
null pointer to struct point
```

This program demonstrates how printing structs works for us. When you print a pointer to a non-null struct, it loops through the fields in order, and prints the runtime values. It also points the address of the struct.

One limitation of our program is that we didn't have time to implement is saving type information; we just print everything as an int; for pointer fields within a struct, you can still compare the address.

When you call print on a struct pointer with value 0x0, ie, null, it displays that the struct you printed is null, as well as the type of the pointer.

`input/bst.snek`

```
% ./tests/input/bst.run
struct bst (4308189680) ; root
        val: int = 4
        left: bst = 0
        right: bst = 0
null pointer to struct bst ; root's left child
null pointer to struct bst ; root's right child
struct bst (4308189680) ; root
        val: int = 4
        left: bst = 4308189704
```

```
        right: bst = 0
struct bst (4308189704) ; root's left child
        val: int = 3
        left: bst = 0
        right: bst = 0
null pointer to struct bst ; root's right child
struct bst (4308189680) ; root
        val: int = 4
        left: bst = 4308189704
        right: bst = 0
struct bst (4308189704) ; root's left child
        val: int = 3
        left: bst = 4308189728
        right: bst = 0
null pointer to struct bst ; root's right child
struct bst (4308189680) ; root
        val: int = 4
        left: bst = 4308189704
        right: bst = 4308189752
struct bst (4308189704) ; root's left child
        val: int = 3
        left: bst = 4308189728
        right: bst = 0
struct bst (4308189752) ; root's right child
        val: int = 5
        left: bst = 0
        right: bst = 0
```

In this example, we start by allocating a tree with value `4` and no children.

Then for each of: `3, 2, 5` (in that order), we insert the new number into the BST and then print the root (which contains 4).

As the trace shows, initally the root has `null` for left and right. Once we insert `3`, since that's smaller than 4 it is inserted as the left child of the root, which is now non null. When we insert `2`, since it's smaller than 4 it belongs to the left subtree, and is inserted as the left child of `3`. When we insert `5`, which is larger than 4, it goes into the right subtree, which is why in the final printout the root's right child is non-null.

`input/error-alloc.snek`

```
% ./tests/input/error-alloc.run
Runtime error: out of space
```

This program calls `alloc` in a `repeat-until-false` loop (which would run forever), except it encounters a runtime exception when our bump-allocator runs out of room to allocate the next struct.

This error happens at runtime, as `lookup` / `update` check for null-pointer dereference.

`input/error-read.snek`

```
% make tests/input/error-read.run
cargo run -- tests/input/error-read.snek tests/input/error-read.s
   Compiling cobra v0.1.0 (/Users/elijahbaraw/Desktop/private/17363/hw/hw4-cobra/17363-
Cobra)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.68s
     Running `target/debug/cobra tests/input/error-read.snek tests/input/error-read.s`
thread 'main' panicked at src/typecheck.rs:638:25:
Invalid: Lookup nonexistent field value in struct ll
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
make: *** [tests/input/error-read.s] Error 101
```

This test demonstrates what happens when a program attempts to access a nonexistent field; it throws a **compile time** error, which identified the struct name, and the field name which isn't in that struct. This error happens at compile time, during type-checking (after parsing, before compiling).

From the declaration of `ll`, it has no field `value` (the correct field is `val`): `(struct ll ((val int) (next ll)))`.

`input/error3.snek`

```
% ./tests/input/error3.run
Runtime error: null dereference
```

In this test, a null pointer is dereferenced at runtime, which generates a runtime exception: `Runtime error: null dereference`.

Comparison to Real Programing Languages

Two languages which supports heap-allocated data are C, and SML. In C, memory management isn't a feature of the language, but is implemented user-side functions, typically `malloc()` and `free()`. The user of the program is responsible for checking if `malloc()` failed, and for only calling `free()` on memory it "owns" (ie, got from a call to `malloc()`). C also allows for pointer arithmetic and reading / writing to arbitrary memory.

Our language, in contrast to C, provided memory management as a *language* feature, and the user can't do pointer arithmetic. So, the only pointer which the user has access to which isn't a pointer to a valid struct is `null`, and we perform runtime checks that we don't dereference `null` for every pointer dereference, which provides us with memory safety.

SML is much closer to our model than C, since SML doesn't expose pointers to the user, and doesn't require explicit allocation & deallocation as C does. In SML, users can create ref cells which are mutable references to a specific. These are very similar to our structs, which are created via `alloc` and are pointers to memory under the hood. Our structs are also typed, so that at compile-time we know how a pointer should be treated.

Our language is different from SML in that we have an allocate-only bump allocator, ie we never free a struct, where as various SML implementations use approaches like garbage collection to automatically de-allocate ref-cells once they are no longer used by the program.

List of Resources

Designing a Bump Allocator: https://cohost.org/eniko/post/171803-basic-memory-allocat

Storing and Accessing a `.bss` Array: https://stackoverflow.com/questions/34058101/referencing-the-contents-of-a-memory-location-x86-addressing-modes
https://www.reddit.com/r/asm/comments/16tdhvh/data_vs_bss_for_uninitialized_data/

Pretty Printing and String Methods: https://stackoverflow.com/questions/70096134/assembly-store-a-string-in-register https://docs.rs/windows-win/latest/windows_win/sys/struct.CStr.html