Archetype: Domain specific language for the representation of Abstract Algebra

INTRODUCTION

We seek to make a language which can be used to represent and manipulate algebraic structures, which we call Archetypes. The language is designed to be used by mathematicians, and so the syntax is designed to be similar to mathematical notation while being concise and easy to learn.

Syntax

Statements

- The language is case sensitive.
- All statements end with a semicolon.

```
let a: u32 ; // declaration
a = 1; // assignment
a = func(2); // function call
```

Types of statements:

- Declaration: `let a: u32;`
- Assignment: `a = 1;`
- Function//Method call: `a = func(2);` or `a = q.func(3);`
- Initialisation: `let a: u32 = 1;`
- Return: `return a;`

Comments

- Single line comments begin with `//`.
- Multi-line comments begin with `/*` and end with `*/`.
- Comments cannot be nested.

Operators

- Relational: >, <, ==
- Logical: &&, ||, !
- Arithmetic: +, *, -, /
- Shifts, etc.

All the operators have the same meaning as in C, with enhanced functionality for non-C types (matrices, for example).

Conditionals

- The keywords `if` and `else` are used as in standard languages. Any members of a group, ring or field may be compared using relational operators (again, the standard >, < and ==) as part of the predicate. Booleans are already a group.
- The body of statements is enclosed in curly braces.
- The syntax:

```
if <pred> {
    <body>
}
else if <pred> {
    <body>
}
else {
    <body>
}
```

Loops

- The `for` keyword is used to iterate over a range of values. The syntax is similar to C.

```
for (declaration; predicate; operation) {
    .
    .
    .
}
```

- Using the `for` and `in` keywords, we can iterate over the members of a vector.

```
for (member in list) {
    .
    .
    .
}
```

Where list is a vector over type `T`, and thus the type of `member` is also `T`.

- The `while` keyword can be used with a predicate as usual.

```
while (predicate) {
    .
    .
    .
}
```

Functions

Function prototypes begin with the `fn` keyword, followed by the function name, the arguments within parentheses, and then the return type.

```
fn <name>  (arg: type, ...) : <return type> {
    <body>
}
```

Functions calls are identical to C: `name(args)`. Functions can be returned from using the `return` keyword, which is identical to C.

Type system

We have devised a rich and flexible type system to aid in expressing complex algebraic concepts.

Structs

Definition:

```
struct name {
    members
}
```

To access a field, use the `.` operator.

```
let u: name; // declaration
u.field = 1; // assignment
```

Enums

```
enum name {
    members
}
```

Use the `::` operator to depict enum variants.

```
let u: name; // declaration
u = name::variant; // assignment
```

Archetypes

Each type, except for the System types, is assigned one or more of the {four}(five if we adding Collection) Archetypes, which are as follows

Group

A group may be defined as an amalgamation of a set and an operation. The operation must satisfy certain bounds. With a set S and an operation $f(a, b) = a.b$:

- Closure: $\forall a, b \in S, a.b \in S$.
- Associativity: $ a.(b.c) = (a.b).c$
- Existence of
  - identity $\exists i \in S | a.i = a \forall a \in S$
  - inverse $\forall a \in S \exists a^{-1} \in S | a.a^{-1} = i$

Thus, a Group may be `claim`'d in our language (see below) by specifying an operation which satisfies these bounds, as well as an identity element and the inverse operation.

Abelian Group

- A group is abelian if it's operator is commutative, i.e., $a.b = b.a \forall a, b \in S$.

Members: ?

Ring

A ring is an abelian group with another operation, $*$. Using the same notation as before, the additional properties of a ring are:

- Closure: $\forall a, b \in S, a * b \in S$
- Associativity: $a * (b * c) = (a * b) * c$
- Distributivity: $a * (b.c) = (a * b).(a * c)$
- Existence of identity: $\exists e \in S | a * e = a \forall a \in S$

Members:

- Unsigned integers `u8`, `u16`, `u32`, and `u64` : Our language does not treat integers and reals as primitive data types.
- `BigInt`: Unbounded integer type, similar to integers in Python.
- `Matrix`
  - A matrix is treated as a generic over any type, but the methods it provides wil depend on the Archetype of that type. For example, a matrix of integers will not have the inverse operation, because the inverse of a matrix of integers may contain reals.
- `Polynomial`
  - Similar to matrices, they can be generic over any type.

Commutative rings

- When the $*$ operation is commutative, the ring is said to be commutative.

Field

A field is a commutative ring with the additional property that every non-zero element has an inverse in the second operation. Using prior notation, $\forall a \in S, a \neq i \Rightarrow \exists a^{-1} \in S | a.a^{-1} = e$.

Members:

- reals
- complex numbers
- BigRational
- Non-Singular matrix (multiple Archetypes)
- Polynomials over a field (multiple multiple Archetypes)

Space

The only member is the `vec` - for vector. It is generic over types that claim `Field` and `Ring`{, although providing different operations for each. }(do they really?).

- Similar to vectors in C++ and Java.
- They are generic over any type, and the operations they provide will depend on the Archetype of that type.
- They provide basic array functionalities such as indexing, appending, etc., but also algebraic vector operations such as adding two arrays together, and scalar multiplication.

```
let a: Vec<u64> = {1, 2, 3}; // initialisation
let b: Vec<u64> = a * 2; // Scalar Multiplication
let c = a + b; // Vector Addition
let c: u64 = a[0]; // Indexing
```

- **Inner products** This is automatically implemented. `let a: Vec<u64> = {1, 2, 3}; let b: Vec<u64> = {4, 5, 6}; let c: u64 = a @ b; // Inner product`

  If the programmer wishes to claim the `Space` Archetype, they must implement the inner product operation themselves.

System type

- These are the data types/objects offered by the system, and while they may be represented using algebraic constructs (aka Archetypes), those structures

are relatively more complex and esoteric. Naturally, the programmer may use these types to build more complex structures.
- Wrapping a `System` type within a `struct` allows the programmer to claim an Archetype for these types, and thus use them in algebraic operations.

Pointers

Pointers are used to refer to objects in memory, in a very similar fashion to C and C++. The syntax is as follows:

```
let a: u32 = 1;
let b: &u32 = &a; // b is a pointer to a
let c: u32 = *b; // c is the value pointed to by b
```

Boolean

Booleans are implemented as `System` types eve though they technically satisfy the definition of a group. This is because they are used in the control flow of the program, and thus are not used in algebraic operations. The associated keywords are `true` and `false`.

```
let a: bool;
a = true;
a = !false;
```

Logical (&&, ||, !) operators work on booleans as expected.

- {Strings}(Consider making Collection Archetype with more than just strings. Maybe other data structures? Might be useful to have arrays without algebraic operations.)
- {Tuples}(? Or is the cartesian product of two Archetypes also an Archetype?)

The claim keyword

To create a new instance of an Archetype, the programmer may use the claim keyword. The syntax is as follows:

```
claim (name is Archetype) {
    implement operations
};
```

And one can check whether a type has claimed an Archetype using

```
if (name is Archetype) {
    .
    .
    .
}
```

where name is the name of a type that has already been declared (struct). The new type may directly implement the operations, or define mappings (morphisms) to some other type which implements the operations, like so:

```
morph (self to other) {
    Function accepting self and returning other
};
```

Kinds of morphisms? Note, this is not inheritance, as the programmer cannot write `claim cat is animal {...}`.

Tokens

Reserved words

- `let`
- `if`
- `else`
- `for`
- `while`
- `fn`
- `claim`
- `is`
- `struct`
- `enum`
- `true`
- `false`

Data types

- `BigInt`
- `Matrix`
- `Polynomial`
- `u8`
- `u16`
- `u32`
- `u64`
- `real`
-