PLATYPUS

Dylan Maloy (dylan.maloy@tufts.edu)
Rodrigo Campos (rodrigo.campos@tufts.edu)
Ronit Sinha (ronit.sinha@tufts.edu)
Tony Huang (ziheng.huang@tufts.edu)
Abe Park (yangsun.park@tufts.edu)

COMP 107, Compilers
Tufts University, Spring 2023

1 Introduction

Platypus is a language designed to combine a safety-first ownership model and a developer friendly syntax. The intention is to enforce memory-safe development practices and perform all necessary memory management at compile time.

Rust, the largest presence in this domain, is a low-level systems language that can be difficult to understand for a user more familiar with higher-level languages such as Python. The goal of Platypus is to act as an intermediary between these two archetypes, providing the best of both worlds for developers who want safe and well-defined memory usage in their programming experience without being bogged down by intricacies and pitfalls of self-managed memory.

The following document contains a brief Language Tutorial and a Language Reference Manual, followed by a discussion of the compilers architecture, development process, and testing mechanisms.

2 Language Tutorial

2.1 Getting Up and Running

We've provided a few different ways to run our compiler. For guaranteed success, a pre-built docker image can be used.

If you've already downloaded the repository, you can use the make commands:

1. `make docker.pull`
2. `make docker.run`

Otherwise, you can use the following shell commands:

1. `docker pull ghcr.io/dolpm/platypus:latest`
2. `docker run -rm -it -v platypus:/app/ -w=/app/ ghcr.io/dolpm/platypus:latest`
3. `make`

If you would like to build the compiler without the docker image, here are a list of dependencies that you will need:

1. `llvm 14.0.6`
2. `clang 14.0.6`
3. `ocaml 4.14.1`
4. `opam 2.1.4`

After these have been installed on your system, clone the git repository:

1. `git clone https://github.com/dolpm/platypus`

Then, after entering the repository directory, run the following commands to complete the installation:

1. `make install`
2. `eval $(opam config env)`
3. `make`

If all was successful, a `platypus` binary should appear in the `platypus` directory. To test the installation, run the command: `make test`.

2.2 Compiling a Platypus Program

To use the `platypus` executable that was generated in the previous step, please refer to our repository README. It delves into the compiler commands and their use-cases, in addition to some of the optional flags that can be provided.

2.3 The Basics of a Platypus Program

2.3.1 Hello, world!

```
1 pipe main [] |> unit {
2     Printnl <| ["Hello, world!"]; /* this is a comment! */
3     |> ();
4 }
```

The starting point of any Platypus program is the `main` pipe. Pipes are Platypus's equivalent to functions in other languages. So Platypus's `main` pipe is like the main function in C. The `main` pipe takes in no arguments and has a return type of `unit` (more on types later, for now just know that the unit type is like C's `void`).

Line 2 is a call to the builtin `Printnl` pipe, which prints its given argument (in this case, the string `"Hello, world!"`), followed by a newline. Also notice how comments in Platypus are written. Then, line 3 is a pipe-out — similar to a function return in other languages. The return value in this case is just the unit literal `()`. In Platypus, every pipe must have a pipe-out; even those that have a unit return type.

When the above code is executed, we get the following output:

```
1 Hello, world!
```

In order for any code to execute, it must be contained within a function (with notable exceptions — more on those later). For the sake of brevity, the code samples in this tutorial are just the bodies of pipes; they don't include the surrounding pipe declaration or a pipe-out. If you wish to run any of samples, you can put their contents inside of a `main` pipe with a pipe-out at the end, like we do in the hello world sample above.

2.3.2 Types

Platypus offers a range of different data types. Let's start with the easy ones.

Primitives Platypus's primitive types should be familiar to anyone with a little programming experience. `int`, `bool`, `char`, `float`, `string`, and `unit` are all types that have analogs in various other languages. The `unit` type is like C's `void` type. It is exclusively used for defining pipes with no return value.

You can create variables by specifying types and assigning them values:

```
1 int i = 1337;
2 float f = 1337.41;
3 string s = "Hello, world!";
4 bool b = true;
```

By default, variables are immutable (like constants in other languages). If you want them to be mutable, they must be defined with the `mut` keyword, like so:

```
1 mut int x = 17;
2 x = 38; /* now we can update x! */
```

Composites Platypus also has composite types, which are data types that depend on other data types. Take vectors for example. Vectors are resizable lists whose type depends on the type of its elements. So the `vector[int]` type is a vector of integers, whereas the `vector[vector[int]]` type represents a vector in which each element is a vector of integers.

There are other composite types (like Things, which are like C's structs), but the one most central to understanding Platypus are *references*.

2.3.3 References, Borrowing and Ownership

References References are like pointers in other languages; they refer to another value. There are two types of references: immutable and mutable. As their names suggest, immutable references can read the value they are referencing, but cannot alter it, whereas mutable references are allowed to change the value they refer to. References have an associated lifetime, which is basically a measure of how long the data it refers to is safely accessible (i.e. how long the referred value is in scope). Below are some examples of references. Notice that reference variables are created with & and ~& operators. First, an immutable reference:

```
int x = 5; /* an integer value, immutable */
&int a = &x; /* an immutable reference to x */
/* the dereference operator (@) gets the value of a reference */
Printnl <| [@a]; /* prints 5 */
```

And also a mutable reference:

```
/*
    a floating-point value, mut keyword means we can mutably
    borrow it or directly update it
*/
mut float y = 12.3;
~&float b = ~&y; /* a mutable reference to y */
/*
    since b is mutable, we can update y through b
    using the dereference operator (@)
*/
@b = 3.1;
Printnl <| [y]; /* would print 3.1 */
```

Immutable reference variables can be reassigned, but only to references that have a lifetime greater than or equal to the previously-assigned reference's lifetime. Note that the reassignment of an immutable borrow does not change its lifetime, only the value it references. For example:

```
int x = 5;
&int a = &x;
{
    int y = 8; /* y in a smaller scope than x */
```

```
    &int b = &y; /* so b's lifetime is smaller than a's */
    /*
        b's lifetime is still constrained to this block,
        but now it references x
    */
    b = a;
}
```

So, what is the point of references, and why are they such a big deal in Platypus? References are part of Platypus's core feature: *the borrowing and ownership model*. This model, also called the borrow-checker, is a compile-time analysis that ensures that all program memory is safely read, mutated, and deallocated.

Ownership So how does the borrow-checker accomplish this? For starters, it requires that any given point in a program, any data has exactly one *owner*. The owner is a variable that, once it falls out of scope, will deallocate the data it owns (if it is on the heap). Whenever data in a program is created, it is bound to a variable; that is the data's inital owner. But ownership can be transferred from one variable to another. Once that happens, the previous owner can no longer be used in the program. Consider the following example:

```
int x = 10; /* at the start. x owns the value 10 */
int y = x; /* ownership transfer! y now owns x's value */
/* after this point, x can no longer be accessed */
```

There are other ways to transfer ownership (such as passing a variable into a function), but in any case it means the old owner can no longer be accessed. Note that if you want to pass a value from one value to another without transferring ownership, you can make a deep-copy of it using the clone operator (`%`).

```
int x = 10;
int y = %x; /* y gets ownership over a new copy of x's value */
/* x and y can both be accessed since they each own a value */
```

Borrowing Let's get back to references. References are able to read and even modify data without taking ownership of it. In other words, it *borrows* the data. But, in order to ensure memory safety, there are limitations on when and what kinds of references can be taken on a value. Namely, in a given scope, if a mutable reference is taken out on a value, no other references can be taken out on them. And the value can only be updated through said mutable borrow, not through the owner.

The reason for this constraint is that if a reference is taken out on a value while another mutable reference is active, the mutable reference may update the value in such a way that the other reference is now accessing unsafe memory. This would be undefined behavior – not good!

So that is a brief overview of Platypus's borrow-checker. For a more in-depth look, refer to the Language Reference Manual.

2.3.4 User-defined pipes

Recall that pipes are Platypus's equivalent of functions. Users can define their own pipes. Pipes are declared at the top level of a program — no pipes inside of pipes. The order in which pipes are defined matters; pipes can only call pipes that are defined above them. To define a pipe, use the `pipe` keyword, followed by the name of the pipe, and then the arguments wrapped in brackets. Finally, we need a right-pipe symbol (|>) followed by the return type of the pipe. With this, we can specify the pipe's body, which must be wrapped in braces. For example:

```
/* adds two integers and returns the result */
pipe add [x : int, y : int] |> int {
    |> x + y;
}
```

Explicit lifetimes There is a special case of pipe declarations. This is the case when a pipe returns a reference. This can only happen when the pipe is returning one of its arguments that is a reference type; for more details on why this is, refer to the Language Reference Manual.

In any case, when a pipe returns one of its reference arguments, the explicit lifetimes of all possible returned references must be indicated in the pipe declaration. These lifetimes are listed between vertical pipes (|) in between the pipe name and the arguments. The lifetimes must be listed in descending order (from largest lifetime to smallest) and the return lifetime must have the smallest of the explicit lifetime. Consider the following example:

```
pipe foo |'a, 'b| [int_one: &'b int, int_two: &'a int] |> &'b int {
    if (@int_one > @int_two) {
        |> int_one;
    }
    |> int_two;
}
```

Explicit lifetime declarations `'a` and `'b` help Platypus reason about the lifetime of a returned reference, which is why they are only used when a reference argument is returned. In the above example, because either argument `int_two` with lifetime `'a` or `int_one` with lifetime `'b` *could* be returned from `foo`, Platypus forces the developer to play it safe; it enforces that the returned value doesn't outlive *int_one*, the possibly-returned reference with the smallest lifetime.

With this in mind, consider the pipe `bar` that calls `foo`.

```
pipe bar [] |> unit {
    int x = 1024;
    {
        int y = 2048;
        &int z = foo <| [&y, &x];
    }
    |> ();
```

```
}
```

Platypus enforces that z's lifetime is equivalent to that of y's. If z were to outlive y, it would be possible for undefined behavior to occur. To show how Platypus enforces this, consider a similar but *invalid* program.

```
pipe bar [] |> unit {
    int x = 1024;
    mut &int z = &x;
    {
        int y = 2048;
        z = foo <| [&y, &x];
    }
    |> ();
}
```

The program will throw an exception because z could outlive foo's return value – its lifetime is larger than that of y, which is dangerous!

```
Fatal error: exception Failure("the variable being bound is outlived by the
   definition to which it is being bound.")
```

2.3.5 Statements

Okay, so now you know how to define your own pipe. But what exactly can we put inside a pipe's body? We'll now look at various statements you can use to specify the functionality of your pipe.

Loop The loop in Platypus's equivalent of a for-loop. Loops must be declared within pipes. The loop declaration must be provided with an inclusive range as well as an identifier that serves as the iterator over said range. For example:

```
loop 0->4 as i {
    Printnl <| [%i];
}
```

This will iterate through all values in the range, producing the output:

```
0
1
2
3
4
```

An optional step size can be added to the iterator, like in the following example:

```
loop 0->4 as (i, 2) {
    Printnl <| [&i];
}
```

This increment i by 2 each iteration, producing the following output:

```
0
2
4
```

Ownership in loops In the above examples, you may have noticed that we clone or borrow the loop iterator whenever we use it (i.e. we pass `%i` and `&i` into `Printnl`). In inside of a loop, you are not allowed to take ownership of the iterator. The intuition behind this is pretty straightforward: if you take ownership of `i` in an iteration, then at the end of the iteration, how can `i` be incremented/decremented?
Similarly, the body of a loop cannot take ownership of anything outside of the loop. Consider the following *invalid* example:

```
int x = 5;

loop 0->4 as i {
    Printnl <| [x];
}
```

So inside of the loop, the call to `Printnl` takes ownership of x. On the first iteration, `x` would have its ownership taken as expected. But on the next iteration, what happens? The `Printnl` call tries to take ownership of `x` again, but it already had its ownership taken in the previous iteration – uh-oh! This produces an error:

```
Fatal error: exception Failure("ownership of x could be taken in a previous
   loop iteration")
```

While loop Platypus's while loops should be familiar to anyone with a little programming knowledge. While loops are declared with a boolean predicate and a body. The while loop will repeatedly execute the code in its body so long as its predicate evaluates to true. For example:

```
mut int i = 0;
while (i < 5) {
    Printnl <| [&i];
    i = i + 1;
}
```

This will iterate through all values in the range, producing the output:

```
0
1
2
3
4
```

Just like loops, nothing in the body of a `while` cannot take ownership of any values outside of the while loop; this includes any variables in the predicate (which is why in the above example, we pass immutable reference `i` into `Printnl`).

If-Else Platypus's if-statements are almost identical to those in a language like C. If-statements are declared with a boolean predicate, a then clause, and an optional else clause. When the predicate evaluates to true, the then clause is evaluated. If the predicate is false and an else clause is provided, then the else clause will be evaluated. For example, the following code will print "Hihidaruma":

```
if (0 < 5) { /* is true */
    Printnl <| ["Hihidaruma"];
}
```

But the code below will output nothing:

```
if (0 > 5) { /* is false */
    Printnl <| ["Hihidaruma"];
}
```

Finally, the following code outputs "Ulgamoth":

```
if (0 > 5) { /* is false */
    Printnl <| ["Hihidaruma"];
} else {
    Printnl <| ["Ulgamoth"];
}
```

Ownership in if-else When an if-statement has a then-clause and an else-clause, we know that only one of these clauses will be evaluated at runtime. Because of this, Platypus can be more lenient with ownership checking between these two clauses. If a variable in the then-clause takes ownership of data from outside of the if statement, this ownership transfer won't affect the else-clause (and vice versa). In other words, in the else clause, the original owner of the data can still be accessed. For example:

```
int x = 5;
if (true) {
    int y = x; /* ownership of x taken in this clause */
} else {
    /* but we can still access x here
        because clauses won't coexist in runtime */
    x;
}
```

That said, if ownership of data outside of the if statement is taken by a variable in *either* clause, then the original owner cannot be accessed after the if statement. This is because Platypus's borrow-checker is a compile-time analysis — it can't tell which clause of an if statement will be evaluated at runtime. Therefore Platypus plays it safe by prohibiting access of a variable if its ownership is taken by either clause of the if statement.

Consider the following *invalid* code, for example:

```
int x = 5;
```

```
if (true) {
    x;
} else {
    int y = x; /* ownership of x taken in else-clause */
}
/* even though else-clause never gets processed in runtime,
    x cannot be accessed here */
x;
```

When this code is run, we get an error:

```
Fatal error: exception Failure("variable x gave ownership to another binding
   and can't be accessed.")
```

2.3.6 User-Defined Things

Platypus supports user-defined collections of variables called `things`. These are similar to structs in C. Thing declarations are top-level, and once one has been defined, it can be used in any pipe, even if the thing is defined below the pipe declaration (in Platypus, all thing declarations are processed before any pipe declaration). Here is an example of a thing declaration and its usage:

```
thing MyThing <| {
    im_an_int : int,
    im_a_bool : bool
}

pipe foo [] |> unit {
    mut MyThing t = MyThing {
      im_an_int: 1,
      im_a_bool: true
    };

    {
        /* immutably borrowing a thing member */
        &bool b = &t[im_a_bool];
    }

    {
        /* mutably borrowing a thing member */
        ~&bool b = ~&t[im_a_bool];
        /* updating the member */
```

```
21          @b = false;
22      }
23
24      |> ();
25  }
```

2.3.7 Builtin pipes

Platypus provides various pre-defined functions for the developer. You've already seen one so far: `Printnl`! But there are many others; for example, `Vector_new` creates a new vector, `Panic` throws an error with a custom message, and `Rng_generate` generates a random integer. For a full listing of all the builtin pipes Platypus has to over, refer to our Language Reference Manual.

2.3.8 Conclusion

So that's a whirlwind-tour of Platypus! Oh, they grow up so fast *sob*! There's a lot to cover, and the borrow-checker can be daunting at first, but you should now have a cursory understanding of the language. Of course, we didn't go into the full depth of everything our language has to offer. For a more in-depth look, the Language Reference Manual details everything Platypus can do in horrifying detail. Best of luck on your Platypus coding journey!

3 Language Reference Manual

3.1 Reading Directions

Prose is written using plain text, such as: "This is an example of prose."

Short snippets of code are represented in this manner:
`mut [int * bool * float] my_tuple = tuple(21, false, 17.38);`

Code listings are displayed as the following:

```
1 int x = 5 + 2 - 3 / 4;
```

Finally, grammar rules are written as:

$$typ ::= \textbf{int} \mid \textbf{float} \mid \textbf{bool} \mid \textbf{unit} \mid \textbf{char} \mid \textbf{string}$$

Where text that is either bolded or non-alphanumeric are terminals, and unbolded text (i.e. typ) is a nonterminal.

3.2 Lexical Elements

3.2.1 Identifiers

Identifiers must begin with a lowercase or uppercase letter. All remaining characters can be either a lowercase or uppercase letter, a number, or an underscore. In grammar rules, identifiers are represented by the terminal **ident**.

3.2.2 Keywords

Keywords are reserved by the language. They are not able to be used outside of their pre-defined contexts, unless enclosed in a string or comment. The words must be exactly as they are written in the table below. And in grammar rules, each word is represented by a terminal of the same name (i.e `mut` is represented by terminal **mut**).

`mut`	`and`	`or`	`as`
`pipe`	`if`	`else`	`loop`
`while`	`char`	`int`	`float`
`bool`	`unit`	`tuple`	`string`
`thing`	`box`	`vector`	`str`

3.2.3 Reserved symbols

The following symbols are reserved by the language. Similar to keywords, these symbols cannot be used outside of their pre-defined contexts, unless in a string or comment. Each of these symbols is represented by a terminal of the same name in grammar rules (i.e. <| is represented by the terminal <|).

```
(  )  {  }  [  ]  ;  ,  :  .  ->  <|  |> +  -  *  %  /
                ~& &  ^ ==  =  <  <=  >  >=
```

3.2.4 Literals

Boolean literals The text `true` or `false` are called boolean literals. In grammar rules, boolean literals will be represented by the terminal **bool_literal**.

Integer literals An uninterrupted series of digits (possibly preceded by a minus sign) is called an integer literal. In grammar rules, integer literals will be represented by the terminal **int_literal**.

Float literals A series of digits (possibly preceded by a minus sign), followed by a decimal point and then more digits is called a float literal. In grammar rules, float literals will be represented by the terminal **float_literal**.

Lifetime literals A single quote (') followed by a series of non-whitespace, non-reserved characters are called lifetime literals. In grammar, rules, they will be represented by the terminal **lt_literal**. More discussion on lifetimes is later introduced in the Borrow-Checking section.

Char literals A single quote (`'`), followed by a single character of any kind (except a single quote), and then another single quote is called a char literal. Char literals will be represented by the terminal **char_literal** in grammar rules.

String literals A double quote (`"`), followed by a series of any characters (except a double quote), and then another double quote, is called a string literal. In grammar rules, string literals will be represented by the terminal **string_literal**.

Unit literals A pair of parenthesis "()" is called a unit literal. In grammar rules, unit literals will be represented by the terminal **unit_literal**.

3.2.5 Separators

To denote the end of a statement, use a semicolon.

3.2.6 Whitespace

Platypus is a free-format language. Tabs, spaces, carriage-returns, and all other whitespace is ignored (it is only used to delimit tokens).

3.2.7 Comments

Anything written between an opening '/*' and a closing '*/' is considered a comment, thus it is ignored. Comments may span multiple lines.

3.3 Data Types

Data types in Platypus are described by the following grammar rule:

$$
\begin{aligned}
typ ::= & \textbf{ int} \mid \textbf{float} \mid \textbf{string} \mid \textbf{char} \mid \textbf{bool} \mid \textbf{unit} \\
& \mid \textbf{tuple[}typ\textbf{]} \\
& \mid \textbf{ident} \\
& \mid \textbf{vector[}typ\textbf{]} \\
& \mid \textbf{box[}typ\textbf{]} \\
& \mid \textbf{str} \\
& \mid \textbf{\&}typ \\
& \mid \textbf{\~{}\&}typ
\end{aligned}
$$

3.3.1 Primitives

Platypus has 6 primitive types and supports literals for all of them.

Type	*Example Literal*	*Description*
`int`	`1024`	32-bit signed integer.
`float`	`3.14`	IEEE-754 floating point number.
`string`	`"hello"`	0 or more characters (excluding double quotations) wrapped in double quotes.
`char`	`'h'`	A character (excluding the single quotation), newline (\n), or carriage return (\r), wrapped in single quotes.
`bool`	`true`	A boolean value, either `true` or `false`.
`unit`	`()`	Can represent only a single value, itself; similar to void in C.

3.3.2 Composite Data Types

Composite data types are data types constructed using other types. For this section, let typ signify a nonterminal that can be replaced with any type, except for (1) an immutable reference or a reference (see below) or (2) a unit. In other words, Platypus does not allow composite types to contain references or units.

3.3.3 Tuples

Tuples are fixed-size, ordered collections of elements. A tuple type is determined by the types of its elements. The type of a tuple is written as:

$$[typ_1 * typ_2 * ... * typ_n]$$

In the above definition, typ_i specifies the type of the ith element. For example, `[int * bool * char]` is a tuple whose first element is an integer, second element is a boolean, and third element is a character. Platypus also allows tuple literals. For a tuple type $[typ_1 * typ_2 * ... * typ_n]$, the tuple literal is

$$\texttt{tuple}(v_1,\ v_2,\ ...\ v_n)$$

Where v_i is a value of type typ_i.

For example, the following is tuple literal of type $[int * bool * char]$:

```
tuple(40, false, 'T');
```

3.3.4 Things

Things are user-defined structure types. Things have a name and a series of name fields, each with a specified type. Their declarations are top-level. The thing declaration is written as follows:

$$\texttt{thing}\ \ t\ \texttt{<|}\ \{\ x_1 : typ_1,\ x_2 : typ_2,\ ...\ x_n : typ_n\ \}$$

This defines a thing type t, so variables of type t may be created. A literal of type t can be written as:

$$t\ \{\ x_1 : v_1,\ \ x_2 : v_2,\ ...\ x_n : v_n\ \}$$

Where v_i is a value with same type as x_i, as defined in t's declaration.

Given the following thing declaration:

```
thing MyThing <| {
    im_an_int : int,
    im_a_bool : bool
}
```

An example of a `MyThing` literal would be:

```
MyThing {
    im_an_int : 1738,
    im_a_bool : true
};
```

3.3.5 Vectors

Vectors are resizable, ordered lists of elements that are all of the same type. A vector type is determined by the type of its elements. The vector type is written as follows:

$$\texttt{vector[}typ\texttt{]}$$

In the above notation, typ specifies the type of the elements. For example, `vector[int]` is a vector of integers, and `vector[vector[int]]` is a vector in which each element is a vector of integers.

3.3.6 Boxes

Boxes are similar to pointers in other languages, as they point to data stored on the heap. A box type is determined by the type of the value it is pointing to. The box type is written as follows:

`box[`*typ*`]`

In the above notation, *typ* specifies the type of the value that the box is pointing to. For example, `box[int]` is a box that points to an int on the heap, and `box[box[int]]` is a box that points to a box, which points to an int.

3.3.7 Str

A Str is Platypus's implementation of a heap-allocated string. Unlike `string` values, which are constants, Strs are variable-size, and their type is written as `str`. There are not `str` literals in Platypus. Instead, Strs are constructed using the `string` type (see Builtins section for more details), making them a composite type.

3.3.8 Immutable References

Immutable references are read-only references to a value, like a read-only pointer. An immutable reference type is determined by (1) a lifetime (see Borrow-Checking section for more details), which is represented by a string of characters starting with a `'`, and (2) the type of the value it references. The immutable reference type is written as follows:

&*lt typ*

In the above definition, *lt* is the lifetime of the reference (it is a **lt_literal**) and *typ* is the value it is referencing. For example, `&'a int` is an immutable reference to an `int` with lifetime `'a`, and `&'b vector[bool]` is an immutable reference to a boolean vector with lifetime `'b`.

3.3.9 Mutable References

Mutable references are similar to immutable references, except they allow you to modify the value they are referencing (i.e. they are not read-only). An mutable reference type is determined by (1) a lifetime (once again, represented by a string of characters starting with a `'`) and (1) the type of the value it references. The mutable reference type is written as follows:

~&*lt typ*

In the above definition, *lt* is the lifetime of the reference (it is a **lt_literal**) and *typ* is the value it is referencing. For example, `~&'a char` is a mutable reference to a `char` with lifetime `'a`.

3.4 Operators

3.4.1 Arithmetic Operators

The binary arithmetic operators take two arguments. They must either both be integers or both be floats. The result type of the operation is the same as the argument types (i.e `float` + `float` = `float`). There is one unary arithmetic operator (negation), which takes one argument of type `int` or `float`. The result type of this unary operation is same as the argument type (i.e. `-int` = `int`). While both arguments must be of the same type, Platypus offers casting functions to ease conversion between the types (see Builtins section for more details).

Operator	*Description*
`+`	*Numeric addition, binary operator.*
`-`	*Numeric Subtraction, binary operator.*
`-`	*Numeric Negation, unary operator.*
`*`	*Numeric multiplication, binary operator.*
`/`	*Division, binary operator. In the case of integers, if the divisor does not evenly divide the dividend, this operation rounds the quotient down to the nearest integer.*

3.4.2 Logical Operators

The binary logical operators take two arguments, both of type `bool`, and return a value of type `bool`. The unary logical operator (negation) takes one `bool` argument and returns a value of type `bool`.

Operator	*Description*
`and`	*Logical AND, binary operator.*
`or`	*Logical OR, binary operator.*
`!`	*Logical NOT, unary operator.*

3.4.3 Comparison Operators

All comparison operators are binary, taking two arguments of the same type. The `==` and `!=` operators take arguments of type `int`, `float`, `bool`, `string`, `str`, or `char` (so long as the arguments have the same type) and return a value of type `bool`. The other comparison operators take arguments of type `int` or `float` (so long as the arguments have the same type), and return a value of type `bool`.

Operator	*Description*
`==`	*Equality*
`!=`	*Inequality*
`>`	*Greater than*
`<`	*Less than*
`>=`	*Greater than or equal to*
`<=`	*Less than or equal to*

3.4.4 Reference Operators

All reference operators are unary, taking a single argument. The `&` and `~&` operators take an argument of any type *typ* other than another than an immutable or mutable reference and return a value of type

&*lt typ* or ~&*lt typ*. The @ operator takes an argument of type &*lt typ* or ~&*lt typ* and returns a value of type *typ*.

Operator	*Description*
`&`	*Borrow, create an immutable reference of the argument*
`~&`	*Mutable borrow, create a mutable reference of the argument*
`@`	*Dereference, get the value pointed to by a reference*

3.4.5 Other Operators

The `%` operator is unary, taking a single argument of type *typ*, where *typ* is not a mutable or immutable reference, and returning a value of the same type *typ*. The `^` operator is binary, taking two arguments of type `str`, and returning a value of type `str`.

Operator	*Description*
`%`	*Deep copy*
`^`	*String concatenation*

3.4.6 Operator Precedence

The precedence of Platypus's operators are defined in the following table. A higher group number corresponds to a higher precedence.

Group	*Operator(s)*	*Associativity*
1	`or`	left-associative
2	`and`	left-associative
3	`==, !=, ^`	left-associative
4	`>, <, >=, <=`	left-associative
5	`+, -` (subtraction)	left-associative
6	`*, /`	left-associative
7	`&, ~&, @, %`	right-associative
8	`!, -` (negation)	right-associative

3.5 Expressions

Expressions have associated types and are evaluated to values. In our language, their syntax can be described by the following grammar rule, which may be later referenced using the *expr* placeholder:

expr ::= *prim_literal* | *comp_literal*
| **ident**
| *thing_access*
| *tuple_index*
| *expr binop expr*
| *unop expr*
| **ident** <| **[** arg_opts **]**
| **(***expr***)**

3.5.1 Primitive Literals

In Platypus, primitive literals are considered *expr*s. They are specified by the following grammar rule:

$$\begin{aligned} \textit{prim\_literal} ::= & \textbf{ int\_literal} \\ & \mid \textbf{float\_literal} \\ & \mid \textbf{bool\_literal} \\ & \mid \textbf{char\_literal} \\ & \mid \textbf{string\_literal} \\ & \mid \textbf{unit\_literal} \end{aligned}$$

The type of primitive literal *expr* is the type of the primitive. So integer literals have type `int`, float literals have type `float`, boolean literals have type `bool`, and so on.

3.5.2 Composite Type Literals

Similarly, composite type literals are also considered *expr*s. They are specified by the following grammar rule:

$$\begin{aligned} \textit{comp\_literal} ::= & \textbf{ tuple (} \textit{ expr}_1 \textbf{ , ... , } \textit{expr}_n \textbf{)} \\ & \mid \textbf{ident \{ ident}_1 \textbf{ : } \textit{expr}_1 \textbf{ , ident}_2 \textbf{ : } \textit{expr}_2 \textbf{ , ... , ident}_m \textbf{ : } \textit{expr}_m \textbf{ \}} \end{aligned}$$

In the first case of this rule, the type of the *expr* will be `[type of` $\textit{expr}_1$ `* ... * type of` $\textit{expr}_2$`]`. For example, `tuple(true, 12)` is an *expr* of type `[bool * int]`.

In the second case of the rule, **ident** is the name of a declaring `thing`. $\textbf{ident}_1$ through $\textbf{ident}_m$ are the fields specified in **ident**'s thing declaration. These fields must appear in the same order in the literal as they do in the thing declaration, otherwise Platypus will raise an error. Each $\textit{expr}_i$ must have the same type as the type of the thing field $\textbf{ident}_i$. Finally, the type of the resulting composite literal *expr* is the thing type **ident**. See the Program Structure section for more information about thing declarations.

3.5.3 Identifiers

Identifiers can be names of variables. A variable name on its own can be an *expr*. The type and value of this *expr* will be the type and value of the variable.

3.5.4 Accessing Tuples and Things

Both the accessing of values inside of a `thing`, and the indexing of values inside of a `tuple` are *expr*s.

Thing access Thing access is described by the following grammar rule:

$$\textit{thing\_access} ::= \textit{expr}\textbf{[ident]}$$

In the above definition, *expr* is a reference (mutable or immutable) to a value with thing type t, and **ident** is the name of one of t's fields, as specified in t's thing declaration (see Data Types section for more information). If **ident** is not the name of one of t's fields, an error is thrown. The resulting type of this access is a reference to the type of the accessed value. If *expr* is a mutable reference, then this thing access reference is mutable, and if *expr* is an immutable reference, it is an immutable reference.

For example, considering the following thing type:

```
thing MyThing <| {
    im_an_int : int,
    im_a_bool : bool
}
```

Then, for an instance of `MyThing` named `x`, `x`'s `im_an_int` member could be mutably borrowed with the *expr* `~&x[im_an_int]`. The resulting value would be of type `~&int`.

Tuple indexing Similarly, tuple indexing is described by the following grammar rule:

$$tuple\_index ::= expr\textbf{[int\_literal]}$$

In the above definition, *expr* is reference (mutable or immmutable) to a tuple. The **int_literal** represents the index of the tuple value to access (note that tuples are zero-indexed). If the **int_literal** is out of bounds of the given tuple's size, then an error is thrown. Otherwise, the resulting type of this tuple index expression is a reference of the type of the accessed value. If *expr* is a mutable reference, then this tuple index reference is mutable, and if *expr* is an immutable reference, it is an immutable reference.

For example, given a tuple *expr* `t` of type `[int * bool]`, the last index could be immutably borrowed with the *expr* `&t[1]`. The resulting value would be of type `&bool`.

Note: Both thing access and tuple indexing *expr*'s can be chained to retrieve nested values. Continuing the above example, if given an *expr* `t` of type `[MyThing * float]` (i.e. *t* is a `tuple`), the *expr* `&t[0][im_a_bool]` could be used to immutably borrow the boolean value inside of `t`'s instance of `MyThing`.

3.5.5 Operations

Platypus considers a binary operator *binop* that is preceeded and followed by an *expr* (i.e., *expr binop expr*) a valid *expr*. It also considers a unary operator *unop* followed by an *expr* (i.e., *unop expr*) a valid *expr*. The resulting type of an operation is dependent on the operator and operand(s). For type specifics, please reference the *Operators* section.

3.5.6 Pipe-ins

Pipe-ins are Platypus's function calls. Once a pipe (pipes are Platypus's equivalent of functions) has been defined in the program, it can be called. The type of a pipe-in expression is the return type of the pipe being called. A pipe-in *expr* can be described with the following grammar:

$$pipe\_in ::= \textbf{ident}\ \texttt{<|}\ \texttt{[}\ expr_1,\ expr_2, ..., expr_n\ \texttt{]}$$

Where n denotes the name of the pipe being called, and each $expr_i$ is an expression whose type matches the corresponding formal type in n's declaration (see Program Structure for more details).

It should be noted that if a pipe-in has a non-unit return type, it must be on the right-hand side of an assignment or a reassignment. In other words, Platypus requires that the return values of functions be bound to a variable.

3.5.7 Parenthesized expressions

Platypus considers `(` *expr* `)` (i.e., an expression wrapped in parentheses) to be a valid *expr*. The type and value of this *expr* are equivalent to it's inner *expr*'s type and value.

3.6 Statements

Statements are important to the program's lexical structure, scope, and control flow. In our language, their syntax can be described by the following grammar rule, which may be later referenced using the *stmt* placeholder:

stmt ::= *expr*;
| { $stmt_1$, $stmt_2$, ..., $stmt_n$ }
| *typ* **ident** = *expr*;
| **mut** *typ* **ident**=*expr*;
| **ident** = *expr*;
| **@ident** = *expr*;
| **if** (*expr*) *stmt*
| **if** (*expr*) *stmt* **else** *stmt*
| **ident** <| [*arg_opts*];
| **loop** *expr* -> *expr* **as ident** *stmt*
| **loop** *expr* -> *expr* **as (ident,** *expr*) *stmt*
| **while** (*expr*) *stmt*
| |> *expr*;

3.6.1 Delimited Expressions

Platypus considers *expr*; (i.e., an *expr* followed by a semicolon) to be a valid statement.

3.6.2 Blocks

Platypus considers { $stmt_1$, $stmt_2$, ..., $stmt_n$ } (i.e., a list of *stmt*s wrapped in curly braces) to be a valid statement. Each block defines a new scope (see Scope section for more details).

3.6.3 Assignments and reassignments

Assignments In Platypus, variables are defined through an assignment of an evaluated *expr* to an **ident**. Assignments come in two forms – mutable and immutable:

mut *typ* **ident** = *expr*;

typ **ident** = *expr*;

If the **mut** keyword is used (i.e., the binding is mutable), **ident** may be rebound or mutably borrowed in subsequent statements. In both cases, semantically-valid programs require that *typ* is equivalent to the type of *expr*. These assignments define a variable named **ident** whose type is *typ*.

Reassignments As stated above, if the original binding is mutable, its associated value can be updated via a reassignment. Reassignments come in two forms – referenced and non-referenced:

@ident = *expr*;

ident = *expr*;

If the variable named **ident** has type of mutable reference, then the first reassignment form is required (referenced reassign). This updates the value that the mutable reference is pointing to, similar to updating a value through a pointer in other languages. In this case, *expr*'s type should be equivalent to the type of whatever the variable **ident** is referencing. For example, if **ident** has type `&'a int`, then *expr* must have type `int`.

If the variable **ident** is any other type, the second reassignment form is required (non-referenced reassign), which simply updates **ident**'s binding to be equal to *expr*. In order to be semantically valid, the type associated with **ident** must be the same as the type of *expr*.

A note to avoid confusion Note that a mutable *binding* (assignment with the `mut` keyword) is different than a mutable *reference* (type ~&*lt typ*). Mutable assignments simply indicate whether or not a variable of any type can be reassigned, whereas a mutable reference is a type itself, denoting a reference with the ability to modify the value it references. The two concepts are related; Mutable references can only be taken out on variables that have mutable bindings, since only mutable bindings can have their values updated. So, for example, consider the following code:

```
1 int x = 3;
2 mut &int y = &x;
```

In the code above, `y` is not a mutable reference; it is an immutable reference with a mutable binding. So `y` can be reassigned to some other immutable reference, but as an immutable reference, it can never update the value it is referencing (in this case, the integer `x`).

3.6.4 If-Else Statements

An if-statements is defined by the following grammar:

if (*expr***)** *stmt*

To be semantically valid, *expr* must be of type `bool`. *stmt* gets evaluated only when *expr* evaluates to `true`.

An if-else statement is defined by the following grammar:

if (*expr***)** *stmt* **else** *stmt*

To be semantically valid, *expr* must be of type `bool`. If the *expr* evaluates to `true`, then the first *stmt* gets evaluated. Otherwise, the second *stmt* (the else case) gets evaluated.

3.6.5 Loops

Playtpus's `loop` statement is similar to for-loops in other languages. Loops have two syntactic forms.

loop *expr* `->` *expr* **as ident** *stmt*

loop *expr* `->` *expr* **as (ident,** *expr***)** *stmt*

All *expr*s in these rules must have type `int`. To start, an integer variable **ident** is defined. The first two *expr*s define the range on which the loop iterates. In each iteration, the integer variable **ident** is set to the current iteration and *stmt* is evaluated. Then, **ident** gets increment by either 1 or, in the second syntactic form, by the value of the third *expr*. Note that the range in Platypus's for-loops is inclusive.

3.6.6 While

Platypus's `while` statements are similar to while-loops in other languages. They are defined by the following grammar:

$$\textbf{while (}expr\textbf{)}\ stmt$$

Where *expr* is of type `bool`. This `while` statement repeatedly evaluates *stmt* so long as *expr* evaluates to `true`.

3.6.7 Pipe-outs

Pipe-outs are Platypus's equivalent of function returns. They are written as:

$$|>\ expr\textbf{;}$$

Where the type of *expr* must be the same as the return type of the pipe this *stmt* is in.

3.7 Program Structure

A Platypus program is contained entirely in one source file. Execution of the program begins at the start of file and continues until it reaches the end of the file. At the top level, Platypus program consists of a list of `pipe` declarations (*pdecl*'s) and a list of `thing` declarations (*tdecl*'s). Some `pipe` declaration *p* can only reference another `pipe` declaration *q* if *q* comes before *p* in the file. Similarly, any `thing` declaration *t* can only reference another `thing` declaration *s* if *s* comes before *t* in the file. But, all **thing** declarations get processes before any `pipe` declarations. This means any pipe can reference any thing, even if the thing is defined below it in the file.

In our language, the programs syntax can be described by the following grammar rule:

$$program ::= program\ tdecl \mid program\ pdecl$$

3.7.1 Thing Declarations

As discussed in the Data Types section, a `thing` declaration is specified by the following grammar rule:

$$tdecl ::= \textbf{thing ident}\ <|\ \textbf{\{ ident}_1\textbf{ :}\ typ_1\textbf{, ident}_2\textbf{ :}\ typ_2\textbf{, ... ident}_n\textbf{ :}\ typ_n\textbf{ \}}$$

In the above rule, **ident** is the name of the newly-defined thing type. each $\textbf{ident}_i$ in $\textbf{ident}_1 ... \textbf{ident}_n$ is a unique identifier. It represents a member of the thing whose type is typ_i.

3.7.2 Pipe Declarations

A `pipe` declaration is specified using the following grammar rule:

$$\begin{aligned} pdecl ::= \textbf{pipe ident} \mid \textbf{lt\_literal}_1\textbf{, ..., lt\_literal}_r \mid\ \textbf{[ident}_1\textbf{ :}\ typ_1\textbf{, ..., ident}_k\textbf{ :}\ typ_k\textbf{]}\ |>\ typ\ \textbf{\{} \\ stmt_1 \\ ... \\ stmt_n \\ \textbf{\}} \end{aligned}$$

In the above rule, the lifetimes (i.e., | **lt_literal**$_1$**, ...,** **lt_literal**$_r$ |) are a list of distinct lifetime literals. If no lifetimes are necessary, the enclosing pipes (i.e., | |) can be omitted from the declaration entirely. Explicit lifetimes are only necessary when the return type of a pipe is a reference; in other words, the pipe is returning one of its arguments that is a reference type (see Borrow-Checking for more details).

The list of identifiers and types (i.e. **ident**$_1$ **:** typ_1**, ...,** **ident**$_k$ **:** typ_k**]**) represent the formals of the pipe; in other words, they represent the arguments that may be passed into these pipes. The formal identifiers must be distinct, Platypus does not allow two formals to share the same name. The type of some formal **ident**$_i$ is typ_i.

3.7.3 Scope

Platypus's scope rules define where variables can be accessed in a program. Global variable declarations are not allowed, meaning variables are defined either as `pipe` formals or through local declarations in the body of a `pipe`. New scopes are created by the following statements:

1. Blocks
2. If-Else statements
3. Loops and while statements
4. Pipe calls (the arguments are in their own scope)

So, for example, variables defined inside a block cannot be accessed outside of that block. Furthermore, the *lifetime* of a reference is defined as the scope of the value it is referencing (see Borrow-Checking section for more details).

3.8 Borrow-Checking

Platypus's semantic analyzer maintains an ownership and borrowing model for compile-time memory management and safety-enforcement.

Similar to the system that ***The Rust Programming Language*** employs, our model enforces that programs are written by the developer such that:

1. no unreachable memory remains allocated
2. no uninitialized memory is accessed

At a high level, we do this by maintaining the following invariants:

1. all initialized memory has a single owner that is responsible for its deallocation
2. any memory can only be mutated by one source — either its owner or a mutable reference — and cannot be mutated while there are other references taken out on it in a given scope.

3.8.1 Ownership

Platypus's ownership and borrowing model ensures that any data in a program has a unique *owner*. The owner of some data is the variable that is responsible for deallocating said data when the variable falls out of scope. Note: since owners deallocate their data and all data must have an owner, Platypus programs are protected from memory leaks and require no manual memory freeing from the developer.

Whenever data is created in a program (i.e. `Vector_new` is called or an instance of a `thing` is created), it must be bound to a variable. This variable is the initial owner of this data. Ownership can be *transferred* from one variable to another in the following ways:

1. Through reassignments. (if `y = x`, `y` now owns `x`'s data)
2. By passing a variable into a pipe-in. (in pipe-in `p <| [x]`, `x`'s data is now owned by `p`'s formal)
3. Through pipe-outs. (if `p` is a one-argument pipe that simply pipes-out its argument, then `y = p <| [x]` means `y` owns `x`'s data).

In practice, the ownership model means that whenever the developer wants to move data from one variable to another, a new copy of the data is not created. Instead, the ownership of the data is transferred, so the new variable has access to the exact same data (same location on stack or heap). If the developer wants a new copy of the data, they must create one using the clone operator (`%`).
For example, with no cloning:

```
int x = 5;
int y = x;
/* x can no longer be accessed because it gave ownership of its value to y */
```

But with cloning:

```
int x = 5;
int y = %x; /* y has a deep-copy of x's data */
/* x can still be accessed because it still owns its data */
```

Once a variable `x` gives ownership of its data to another variable `y`, `x` can no longer be accessed. If `x` was still allowed to be accessed, then `x` and `y` would be two points from which the same data could potentially be mutated, which violates Platypus's borrowing and ownership invariants.

A note on ownership transfers with loops Variables inside of a loop cannot take ownership of data from outside of the loop. This is because during runtime, the value outside of the loop may have had its ownership take in a previous loop iteration, so it may no longer have any ownership to give. For example, the following code is not allowed:

```
int x = 5;
loop 0->10 as i {
    /* not allowed, y takes ownership of x on i = 0, */
    /*  so on i = 1 this is not possible */
    int y = x;
}
```

A note on ownership transfers with if-else statements An `if` statement with a then-clause and an else-clause introduces two scopes (one for each clause) that are mutually exclusive; when the program runs, only one of these clauses will be processed. Knowing this, Platypus is more lenient about ownership-checking between these two scopes. If a variable in the then-clause takes ownership of data from outside of the `if` statement, this ownership transfer won't affect the else-clause (and vice versa). In other words, in the else clause, the original owner of the data can still be accessed. For example:

```
int x = 5;
if (true) {
    int y = x; /* ownership of x taken in this clause */
} else {
    /* but we can still access x here
        because clauses won't coexist in runtime */
    x;
}
```

That said, if ownership of data outside of the `if` statement is taken by a variable in *either* clause, then the original owner cannot be accessed after the `if` statement. This is because Platypus's borrow-checker is a compile-time analysis — it cannot tell which clause of an if statement will be evaluated at runtime. Therefore Platypus plays it safe by prohibiting access of a variable if its ownership is taken by either clause of the if statement. For example:

```
int x = 5;
if (true) {
    x;
} else {
    int y = x; /* ownership of x taken in else-clause */
}
/* even though else-clause never gets processed in runtime,
    x cannot be accessed here */
```

3.8.2 Borrowing

Borrowing is the act of taking out a reference on a value. The developer can create a reference using the borrow or mutable borrow operators described in the Operators section. References are similar to pointers from other languages; they allow either immutable or mutable access to a location in memory without taking ownership of the "borrowed" value.

There are two key rules that Platypus imposes on the developer with respect to borrowing:

1. immutable borrows of a value *can* overlap with other immutable borrows of the same value.
2. mutable borrows of a value *cannot overlap* with other mutable or immutable borrows of the same value.

The reference data types contain a parameter called the *lifetime*. The lifetime of a reference describes the largest scope in which it can be safely accessed (i.e., the scope of the value it is referencing).

Explicit vs. implicit lifetimes In most cases, the lifetime of a reference can be implicit (represented by the string `'_`), in which case the developer doesn't have to specify it since Platypus can infer it at compile-time. But there is one situation where the developer must explicitly specify a lifetime when creating a reference:

Consider the case where a pipe is declared such that it has a return type of reference. In this case, the pipe must return one of its formals. This is because any reference created in the pipe body will be popped off of the stack when the pipe returns, so if such a reference was returned from the pipe, it would accessing unsafe memory.

So, if a pipe is declared that returns a reference, then any pipe formal that the pipe may return must have an explicit lifetime. Furthermore, the return type of the pipe must have the smallest lifetime of the explicit formal lifetimes. Note that when declaring a pipe whose arguments have explicit lifetimes, the lifetimes must be listed in descending, i.e. the rightmost lifetime represents the smallest scope. For example:

```
pipe example |'a, 'b | [x : & int, y : &'b int, z : &'a int] |> &'b int {
    if (@x < @y) {
        |> z;
    }
    |> y;
}
```

In the above declaration of pipe `example`, the lifetime `'b` is smaller than `'a`. So, the return type of `example` is a reference with lifetime of `'b`, the same lifetime as formal `y`. Also, note that no lifetime is specified for `x` since it is never returned.

Platypus requires that a pipe's returned reference have the smallest possible lifetime of its formals to ensure memory safety. In the above example, if Platypus allowed the return reference to have a lifetime of `'a`, then during runtime, if `y` gets returned by the pipe, then this reference `y` will have a lifetime larger than the value it is accessing (which was represented by lifetime `'b`). This reference would be allowed to access data after it had fallen out of scope, which is unsafe.

Reassignments of immutable references An immutable reference `x` can be reassigned to another immutable reference `y` so long as the scope of the value that `x` references is *smaller than or equal to* the scope of the value that `y` references. In other words, `x`'s lifetime must be less than or equal to `y`'s lifetime. The reassignment of a immutable reference does not change its lifetime, only the value it is referencing.

For example, the following code is valid:

```
int b = 5;
&int y = &b;
{
    int a = 3;
    mut &int x = &a;
    /* even after reassign,
        x's lifetime is still constrained to this block,
```

```
        but now it references b */
    x = y;
}
```

because x's origin a outlives y's origin b.

However, the following code will cause an error:

```
int a = 3;
mut &int x = &a;
{
    int b = 5;
    &int y = &b;
    x = y;
}
```

because x's origin a outlives y's origin b.

Reassignments of mutable references Reassignments of mutable references is not allowed.

Reassignments when references are active A variable of any type cannot reassigned so long as there are references to the variable in scope, since that would violate Platypus's borrowing and ownership invariants.

Mutable reference shadowing If a mutable reference bound to some name *a* is passed into a pipe that returns that same reference, which is then bound to some name *b*, then *a* can no longer be used so long as *b* is in scope. This follows the invariant that data can only be accessed by one mutable borrow at a time. So, for example:

```
mut int x = 5;
~&int a = ~&x;
~&int b = return_argument <| [a]; /* this pipe returns the passed in mut ref */
/* a can no longer be used here, but b can be */
```

3.9 Builtin Pipes

The following pipes are not implemented in the Platypus language itself, but provide basic functionality to the developer.

3.9.1 Printing

There are two printing pipes in Platypus: `Printnl` and `Print`. They both take a single argument and have a return type of `unit`. `Print` prints its argument to the terminal, while `Printnl` prints its argument followed by a newline. The type of the argument can be any of the following:

1. `int`, `bool`, `float`, or `string`

2. A mutable or immutable reference to an `int`, `bool`, `float`, `string`, or `str`

Note that the printing pipes only accept a reference of a `str` as opposed to a `str` itself because `str`'s are heap objects, so the printing pipes will not take ownership of it and be responsible for it's deallocation. See the Borrow-Checking section for more information on this.

3.9.2 Randomness

Platypus provides pipes for generating pseudo-random integers. The first is `Rng_init`, which takes a single integer argument and has a return type of `unit`. This pipe initializes the seed of the random number generator based on the integer argument. If the argument is positive, it becomes the seed of the random number generator. If it's negative, the seed is set to the current system time.

The second randomness pipe is `Rng_generate`, which takes in two integer arguments and has a return type of `int`. This pipe generates a random number in the range specified by the arguments (the first argument is the minimum, and the second is the maximum). The random number is generated according to the ISO C99 standard. [1]

Note that it is not required to call `Rng_init` before generating random numbers with `Rng_generate`, but it is recommended. Without calling `Rng_init`, the random seed defaults to 1.

3.9.3 Panic

Panics are Platypus's way of raising errors. The `Panic` pipe takes in a single argument and has a return type of `unit`. The type of the argument can be any of the argument types specified for the printing pipes. When `Panic` is called, it prints its argument on a newline (preceded by the string "Panic! ") and then exits the program with exit code 1.

3.9.4 Boxes

There are multiple pipes in Platypus for creating and accessing a `box`. Their functionality and usage is described below.

3.9.5 Box_new

The `Box_new` pipe takes a single argument of type typ, where typ can be any type that is not a reference (immutable or mutable). `Box_new`'s return type is `box[`typ`]`. This pipe takes in a value of a non-reference type and returns a box that contains the value. In other words, `Box_new` puts its argument on the heap and returns a box that points to it.

Box_unbox `Box_unbox` takes a single argument of `&box[`typ`]`, where typ is any non-reference type. It then returns a value of type &typ, an immutable reference to a value of type typ. `Box_unbox` takes in a box and returns an immutable reference to the value contained inside the box.

Box_unbox_mut Similar to `Box_unbox`, `Box_unbox_mut` takes a single argument of `~&box[`typ`]`, where typ is any non-reference type. It then returns a value of type ~&typ, a mutable reference to a value of type typ. In other words, `Box_unbox_mut` takes in a box and returns a mutable reference to the value contained inside the box. This mutable reference allows the developer to update the value inside of a box.

[1] https://www.dii.uchile.cl/ daespino/files/Iso_C_1999_definition.pdf , under "Pseudo-random sequence generation functions"

3.9.6 Vectors

Platypus provides various pipes for creating, accessing, and modifying a `vector`. Their functionality and usage is described below.

Vector_new The `Vector_new` pipe takes in no arguments and returns a value of type `vector[`*typ*`]`, where *typ* is any non-reference type (Platypus does not allow vectors that contain references, immutable or mutable). The inner type of vector is determined at compile time.

For example, given the following code

```
1 vector[int] v1 = Vector_new <| [];
```

`Vector_new` returns a `vector[int]`.

But in the this code:

```
1 vector[bool] v2 = Vector_new <| [];
```

`Vector_new` returns a `vector[bool]`.

Vector_length `Vector_length` takes a single argument, an immutable reference to a `vector`, and returns an integer representing its length.

Vector_get `Vector_get` allows the developer to access an element of a `vector` at a specified index. The pipe takes two arguments. The first argument is an immutable reference to a `vector[`*typ*`]`. The second is the `int` index to be accessed. The pipe returns an immutable reference of *typ*, which represents the value at the index. If there exists no value at said index (i.e., out of bounds), the program will panic to prevent the accessing of out-of-bounds memory.

Vector_get_mut `Vector_get_mut` allows the developer to mutably access an element of a `vector` at a specified index. The pipe takes two arguments. The first is argument is a mutable reference to a `vector[`*typ*`]`. The second is the `int` index to be accessed. The pipe returns a mutable reference of *typ*, which represents the value at the index. If there exists no value at said index (i.e., out of bounds), the program will panic to prevent the accessing of out-of-bounds memory.

Vector_update `Vector_update` allows the developer to directly update an element of a `vector` at a specified index. The pipes takes three arguments. The first argument is a mutable reference to a `vector`. The second argument is an `int` representing the index to be updated. The third argument is the value that will be set. The type of this third argument is equivalent to the inner-type of the `vector` (i.e., calling the pipe with a first argument of type `~&vector[`*typ*`]`, the third argument must be of type *typ*). `Vector_update` has the return type `unit`.

Vector_push `Vector_push` allows the developer to append an element to a `vector`. The pipes takes two arguments. The first argument is a mutable reference to a `vector`. The second argument is the value to be appended. The type of this second argument is equivalent to the inner-type of the `vector` (i.e., calling the pipe with a first argument of type `~&vector[`*typ*`]`, the second argument must be of type *typ*). `Vector_push` has the return type `unit`.

Vector_pop `Vector_pop` allows the developer to remove the last element of a `vector`. The pipes takes a single argument, a mutable reference to a `vector`. If there is no value to be removed (i.e., there are no items in the vector), then the program will panic. `Vector_pop` has the return type `unit`.

3.9.7 Strs

Platypus provides a few pipes to create and mutate Strs (i.e., variable-size strings). Their functionality and usage is described below.

Str_new The `Str_new` pipe takes a single argument, a string constant (i.e., a value of type `string`), and returns a heap-allocated value of type `str`.

Str_push The `Str_push` pipe takes two arguments. The first argument is a mutable reference of type `str`, and the second is a value of type `char`. The `char` value will be appended to the mutably-referenced `str`. `Str_push` has the return type `unit`.

3.9.8 Typecasting

Platypus provides functions for casting values of certain types to other types.

Int The `Int` pipe takes in an argument of the following types: `float, bool, char, int` and creates an `int` value from it.

Calling `Int` on an int simply returns the same `int`. Calling `Int` on a float floors the value. Calling `Int` on a `bool` produces 1 if the `bool` evaluates to true and 0 if it evaluates to false. Calling the `Int` on a `char` will return the ASCII value of the character.

Float The `Float` pipe takes in an argument of the following types: `float, bool, char, int` and creates a `float` value from it.

Calling `Float` on a `float` simply returns the same `float`. Calling `Float` on an `int` just returns the same value as a float (i.e. 10 becomes 10.0). Calling `Float` on a `bool` produces 1 if the `bool` evaluates to true and 0 if it evaluates to false. Calling the `Float` on a `char` will return the ASCII value of the character as a float.

Str The `Str` pipe takes in an argument of the following types: `float, bool, char, int` and creates a `str` value from it. For any argument, it simply creates `str` whose value is a string of the argument's value.

3.10 Keyword Glossary

Platypus introduces a lot of syntax and symbols surrounding its borrow-checker. This can be daunting to developers who are unfamiliar with the concept of a borrow-and-ownership model to begin with. Below are simplified explanations of various borrow-checker components to help alleviate any confusion.

`mut` By default, all variables in Platypus are immutable. If the developers wants to make a variable that can be updated, they need the `mut` keyword.

`&` Immutable reference. Think of this as a read-only pointer. This token is used in two places: to define an immutable reference type (i.e. `int`) and as an operator to take an immutable borrow of a value.

`~&` Mutable reference. This is like a normal pointer in a language like C. You value can read a value from it or update a value through it. This token is used in two places: to define a mutable reference type (i.e. ~&int) and as an operator to take a mutable borrow of a value.

Some rules with references:

1. Can have as many immutable references to a single variable as you want, but if there are any immutable references in scope, you can't update the value.
2. If you have an existing reference on a variable, you cannot take a mutable reference of it.
3. If there is an existing mutable reference on a variable, you cannot take any more references on that variable, immutable or mutable.

`%` Clone operator. This operator creates a deep copy of a value. When passing a value into pipe that doesn't take a reference, you'll often use clone to make sure you aren't giving away ownership of the original variable to the pipe.

`@` Dereference operator. This is operator is used on references. Think of it like dereferencing a pointer. On a mutable references, you can update the referenced value using this operator (i.e. if `x` is a mutable borrow of `int y`, can update `y` like `@x = 2`). On immutable references, you can only dereference them to read their value.

`<|` Pipe-in. This token comes before a list of arguments that are passed into a pipe. The arguments are enclosed in brackets.

For more concrete examples of how Platypus works, look at the examples on GitHub, like tic-tac-toe, merge_sort, and num_islands.

3.11 Sample Code

Below are a few code-samples to demonstrate how Platypus is used.

3.11.1 Fibonacci Number Generation

The following code calculates and prints the 30th Fibonacci number.

```
pipe fib [i: int] |> int {
  if (i <= 1) {
    |> i;
  }
  |> fib <| [i - 1] + fib <| [i - 2];
}

pipe main [] |> unit {
```

```
  int value = 30;
  int result = fib <| [%value];

  Print <| ["result of fib on input "];
  Print <| [value];
  Print <| [": "];

  |> Printnl <| [result];
}
```

3.11.2 Merge Sort

The following code creates a randomized integer vector of size 10 and sorts the list using the merge sort algorithm. After the sort has completed, it then validates that the list is in fact sorted from least to greatest.

```
pipe print_vec [arr: &vector[int]] |> unit {
  int v_len = Vector_length <| [arr];
  Print <| ["["];
  loop 0->(v_len - 1) as i {
    Print <| [Vector_get <| [arr, %i]];
    if (i != v_len - 1) {
      Print <| [","];
    }
  }
  Printnl <| ["]"];
  |> ();
}

pipe merge [arr: ~&vector[int], l: int, m: int, r: int] |> unit {
  int n1 = (m - l) + 1;
  int n2 = r - m;

  mut vector[int] left = Vector_new <| [];
  mut vector[int] right = Vector_new <| [];

  loop 0->(n1 - 1) as i {
    Vector_push <| [~&left, %@(Vector_get_mut <| [arr, l + i])];
  }

```

```
  loop 0->(n2 - 1) as i {
    Vector_push <| [~&right, %@(Vector_get_mut <| [arr, m + i + 1])];
  }

  mut int i = 0;
  mut int j = 0;
  mut int k = l;

  while (i < n1 and j < n2) {
    &int l_item = Vector_get <| [&left, %i];
    &int r_item = Vector_get <| [&right, %j];

    if (@l_item < @r_item) {
      Vector_update <| [arr, %k, %@l_item];
      i = i + 1;
    } else {
      Vector_update <| [arr, %k, %@r_item];
      j = j + 1;
    }

    k = k + 1;
  }

  while (i < n1) {
    Vector_update <| [arr, %k, %@(Vector_get <| [&left, %i])];
    i = i + 1;
    k = k + 1;
  }

  while (j < n2) {
    Vector_update <| [arr, %k, %@(Vector_get <| [&right, %j])];
    j = j + 1;
    k = k + 1;
  }

  |> ();
}

```

```
pipe merge_sort [arr: ~&vector[int], l: int, r: int] |> unit {
  if (l < r) {
    int m = l + (r - l) / 2;
    merge_sort <| [arr, %l, %m];
    merge_sort <| [arr, m + 1, %r];
    merge <| [arr, l, m, r];
  }
  |> ();
}

pipe validate_sort [arr: &vector[int]] |> bool {
  loop 1->((Vector_length <| [arr]) - 1) as i {
    if (@(Vector_get <| [arr, i-1]) > @(Vector_get <| [arr, %i])) {
      |> false;
    }
  }
  |> true;
}

pipe create_vec_of_size [size: int] |> vector[int] {
  mut vector[int] nums = Vector_new <| [];
  loop 0->(size-1) as i {
    Vector_push <| [~&nums, Rng_generate <| [0, 100000]];
  }
  |> nums;
}

pipe main [] |> unit {
  /* initialize rng with random seed (i.e., anything < 0) */
  Rng_init <| [-1];

  int len = 10;
  mut vector[int] nums = create_vec_of_size <| [%len];

  Print <| ["Before: "];
  print_vec <| [&nums];

  merge_sort <| [~&nums, 0, len - 1];
```

```
101
102   if (validate_sort <| [&nums]) {
103   Print <| ["After: "];
104   print_vec <| [&nums];
105   } else {
106     Panic <| ["Integer sort was unsuccessful."];
107   }
108
109   |> ();
110 }
```

4 Project Plan

4.1 Planning, Specification, and Development

From the beginning of the project, we knew that we wanted to integrate some of the memory-safety features that the Rust compiler employs. This would require the language to be imperative, including features such as *lifetimes*, *boxes*, and *references*. With these basic building blocks, we were able to build a crude version of the compiler's current syntax.

After laying out the languages' syntactic foundation, we were able to start working on functionality! We thought it was best to get started on the *borrow checker* as early as possible, as this area had the most unknowns associated with it. With that said, additional semantic analysis and codegen development began soon thereafter.

Thanks to our early start, we were able to spend the last few weeks ironing out language bugs and tidying up the environment. This also made writing the final report a bit more straightforward because our language was fully complete by this time.

4.2 Timeline

Date	Milestone
Jan 23	Project Brainstorm Begin
Feb 1	Project Proposal Complete
Feb 4	Scanner & Parser Development Begins
Feb 22	Testing Automation Groundwork Complete
Feb 23	Scanner & Parser Complete
Feb 27	Language Reference Manual Complete
Mar 2	Semantic Analysis Development Begins
Mar 7	Borrow Checker Development Begins
Mar 25	Testing Automation Improvements Complete
Mar 25	Codegen Development Begins
Mar 29	Hello World Complete
Apr 2	Vector Development Begins
Apr 6	Vector Development Complete
Apr 11	Auto-Free Development Begins
Apr 11	Thing Development Complete
Apr 15	Cloning Development Begins
Apr 16	Auto-Free Development Complete
Apr 17	Cloning Development Complete
Apr 18	Syntax Overhaul Complete
Apr 18	Tuple Indexing + Thing Access Complete
Apr 19	Extended Testsuite Complete
Apr 26	Final Report Begin
May 2	Docker Image Complete
May 2	Codegen Complete
May 3	Test Suite Complete
May 4	Borrow Checker Complete
May 5	Final Deliverables Complete

4.3 Roles & Responsibilities

- Project Proposal → Dylan, Ronit, Abe, Tony, Rodrigo
- LRM → Dylan, Ronit, Abe, Tony, Rodrigo
- Scanner → Dylan, Ronit, Abe, Tony, Rodrigo
- Parser → Dylan, Ronit, Abe, Tony, Rodrigo
- Types → Dylan, Ronit, Tony
- Borrow Checker → Dylan, Ronit
- Things → Dylan, Ronit, Tony
- Tuples + Boxes + Strs → Dylan, Ronit
- Cloning → Dylan, Ronit
- Frees → Dylan, Ronit
- Built-ins → Dylan, Ronit, Rodrigo
- Environment → Dylan
- Testing → Dylan, Ronit, Abe, Tony, Rodrigo
- Final Report Deliverables → Dylan, Ronit, Abe, Tony, Rodrigo

4.4 Environment + Development Tools

If you are curious about the environment requirements required to compile Platypus, refer to the **Getting Up and Running** subsection of our language tutorial. Speaking of which, we've created a Docker image so that a stable environment is always available independent of the machine that the language was being developed on. This was extremely helpful for running valgrind, which isn't available on M1 macs, and fighting WSL's quirks.

At its core, our OCaml project used the Dune build system; we found this to be far more effective than ocamlbuild. We also used C for aiding with *some* of the builtins (e.g., vectors, variable-size strings, and rng). In addition, we used bash for test automation.

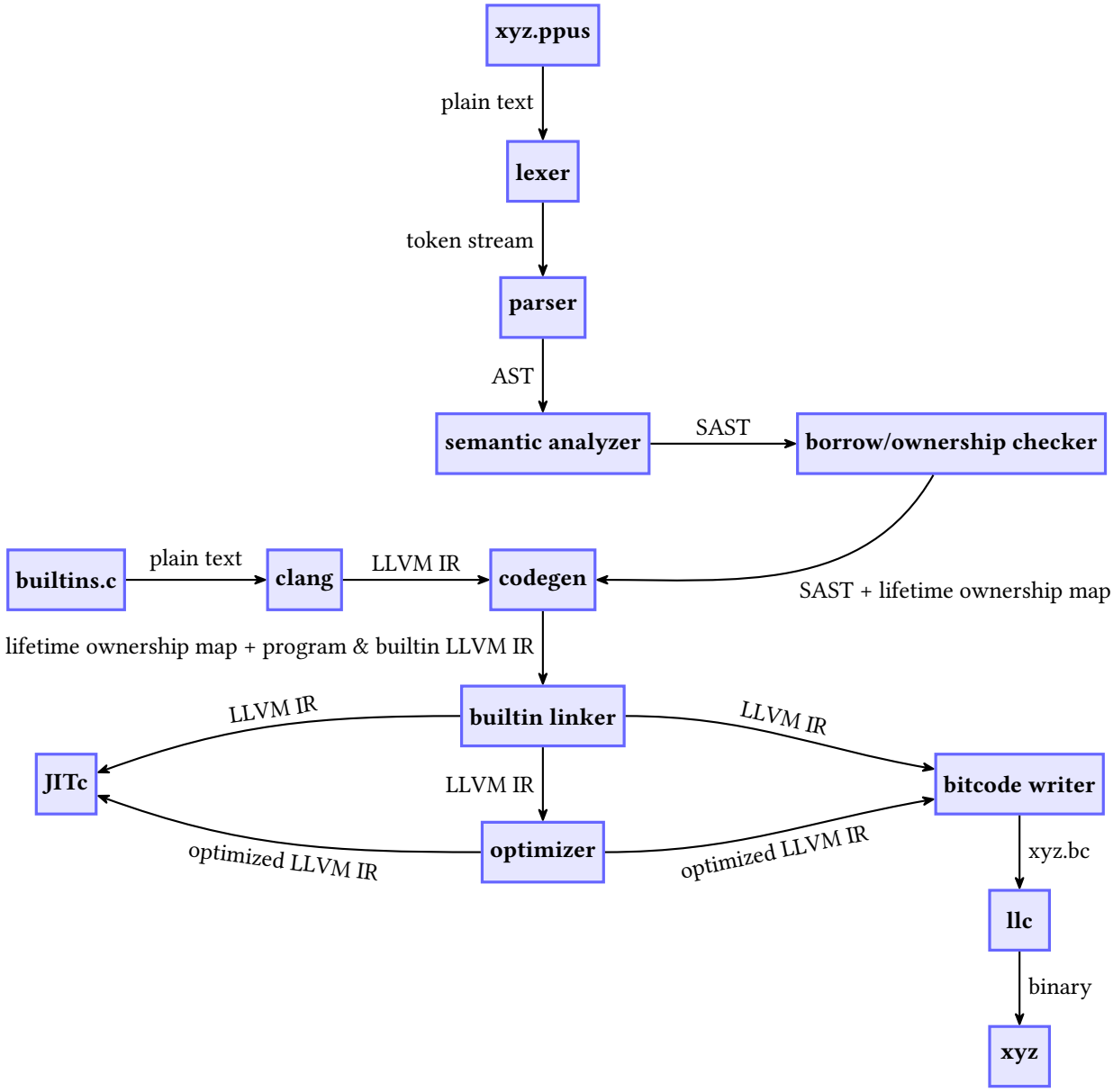
In terms of development environment, we usually used VSCode. To make things easier on the eyes, Dylan developed a syntax-highlighting package for the language.

5 Architectural Design

Platypus's architecture contains all of the basic stages of a compiler:

1. scanning
2. parsing
3. semantics
4. codegen

With that said, there are areas where Platypus differs from other compiler implementations. Below, we have given a more fine-grained graphic description of the workflow:



5.1 Syntax

This first phase of the compiler determines that inputted code is of correctly formed (i.e., that it follows the grammars rules described in the LRM).

5.1.1 Lexing

The first step of the Platypus compiler is scanning, which is done by the lexer. The lexer takes in the raw text from a source Platypus file and parses it into a stream of tokens, which is then sent to the parser.

5.1.2 Parsing

After the program has been tokenized by the lexer, these tokens must be parsed into an *abstract syntax tree* (AST). This is the job of the *parser*, which also enforces that the Platypus code is syntactically valid. If it is not, errors will be thrown in this stage of compilation. The parser returns the generated AST.

5.2 Semantics

This stage of compilation ensures that the code is meaningful (i.e., it follows our semantic and borrow-checking invariants).

5.2.1 Semantic Analyzer

The AST generated by the parser is sent to the semantic analyzer to ensure that it is semantically valid. This involves many different checks. For example:

1. All code adheres to typing rules (can only assign an `int` to an `int` variable, types of arguments passed into a pipe agree with pipe declaration, etc).
2. Variables and pipes are declared before they are used (and for variables, their definitions are in scope when they are used).
3. No pipe declarations have the same name, and in any given scope, no two variable declarations have the same name.
4. No unreachable code (i.e. code after a pipe-out)

And so on. If any of these semantic checks fail, an error will be thrown. Otherwise, a *semantically-checked AST* (SAST) is generated. This data structure is similar to an AST, but contains additional information, such as each `expr` node having an associated type. The generated SAST is then sent to the borrow/ownership checker (see section below).

5.2.2 Borrow/Ownership Checker

Provided the SAST, the borrow/ownership checker ensures that it follows the invariants of Platypus's borrowing and ownership model, as described in the LRM. Some of the things this stage checks for are:

1. If a variable gives ownership of its data to another variable, it is not used after that point.
2. If an immutable reference variable is re-assigned, then the new reference must have a lifetime greater than or equal to the lifetime of the previously-assigned reference.
3. If a variable is already immutably borrowed (i.e., has an immutable reference taken on it), it can't be mutably borrowed.
4. If a variable is already mutably borrowed, it can't be immutably borrowed.

5. If a mutable binding/reference has been shadowed by a more recent mutable borrow on the origin data, it can't be accessed.

6. Lifetime resolution for pipe-calls to pipes that return referenced arguments.

If the source program violates any of the borrow-owning invariants, an error is thrown at this stage of compilation. Otherwise, a lifetime-ownership map is generated. The map is a data structure that keeps track of which variables have ownership of their data at the end of a given scope; this is later used to determined where certain variables should be deallocated. Then, the SAST and the lifetime-ownership map are sent to the LLVM generation phase of compilation.

5.3 LLVM Generation

This phase of the compiler is responsible for generating the appropriate LLVM code that captures the functionality of the inputted Platypus code.

5.3.1 Codegen

The codegen step takes 3 inputs. The first two are from the borrow checker, those being the SAST and the lifetime ownership map. The third input is the LLVM IR of the `builtins.c` file. This file contains some of Platypus's types and builtin pipes, implemented in C. The LLVM of this C file is generated by `clang`, and is then loaded into an OCaml `Llvm.module`, which is the third input to this compilation step.

The codegen compilation step is responsible for generating LLVM code from the SAST. As a part of this LLVM code generation, the lifetime-ownership map is used to generate LLVM instructions that free heap-allocated objects correctly. The lifetime-ownership map keeps track of which variables are owned in a given scope. Since owners are responsible for deallocating their own data (per our Borrow Checking invariants), we know at the end of any scope we must build instructions that deallocate the data of each owner in this scope.

Finally, as we generate code from the SAST, the LLVM code may rely on types from the builtins LLVM (i.e. `Vector`), so we make sure to access them from the builtins' `Llvm.module`.

5.3.2 Builtin Linker

After the codegen has created a LLVM module for the provided `.ppus` code, the compiler makes use of the LLVM linker to link this module with the one that was generated for the builtins by `clang`. This way, builtin pipes and types can be correctly used by the inputted Platypus code.

5.3.3 Optimizer

If the `-o` flag is used in conjunction with the `-e` or `-c` compiler commands, the Platypus compiler will attempt to optimize its generated LLVM IR. It does so by making use of the OCaml `Llvm.PassManager`. The following optimization settings, whose setters can be found here, are applied:

- opt_level: 3
- size_level: 1
- disable_unit_at_a_time: false

- disable_unroll_loops: false
- use_inliner_with_threshold: 10
- internalize: true
- run_inliner: true

5.4 Output

Once the code has been generated by the LLVM Generation phase, Platypus offers multiple ways to "output" the compiled result.

5.4.1 Executable

The compiled Platypus code can be stored as an executable binary. This is accomplished by generating a bitcode file (`.bc`) using the OCaml `Llvm_bitwriter`. This bitcode file is then processed into an executable binary using `llc`. Note that the final output is the executable binary; the temporary bitcode file is deleted after the binary is made.

5.4.2 JITc

Platypus has a just-in-time compiler (JITc) which uses the OCaml `Llvm_executionengine`, which will simply run the compiled code without generating any output files. This option is best if you want to quickly run a Platypus script without writing to disk.

6 Test Plan

6.1 Methodology

Our group approached testing on a feature-by-feature basis. If we were to implement a feature (i.e., the `Printnl` pipe), we tried to have at least one positive integration test showing that the feature functioned as it should. Same went for negative tests – if we threw an error during semantic analysis, we tried to have at least 1 negative and one positive test (a case where the error would be thrown and another where it wouldn't be). It wasn't rare that we would think of an edge case (e.g., a possible flaw with our borrow checker), and would build a feature around this test. For tests regarding scanning/parsing, refer to the **ast** subdirectory in the **test_cases** directory. For tests regarding semantic analysis and borrow checking, refer to the sast subdirectory. For integration tests, refer to the **compile** subdirectory.

6.2 Automation

For more information on how test-running is automated, please refer to our repository README. Our test runner can be found in **test/test.sh**. Most tests are ran though the Platypus JITc to prevent I/O usage, however, memory leak checks (envoked using the memcheck=true argument) for integration tests are done using *valgrind* which requires the full compilation to an executable. All files that are generated during the check will be cleaned up by the runner.

6.3 Example Source + Target Programs

Note: The LLVM code generated for our builtins via clang can be found in the report **appendix**.

6.3.1 pos_str_push – testing the functionality of the Str_push builtin

Listing 1: Source

```
pipe main [] |> unit {
    str h = Str_new <| ["hello"];
    str w = Str_new <| [" world"];

    mut str concatted = h ^ w;
    ~&str c_ref = ~&concatted;

    Printnl <| [c_ref];
    Str_push <| [c_ref, '!'];
    Printnl <| [c_ref];
    |> ();
}
```

Listing 2: Output

```
hello world
```

```
hello world!
```

Listing 3: Target

```
; ModuleID = 'Platypus'
source_filename = "Platypus"

%struct.Vector = type { i32, i32, i8* }

@fmt_str_nnl = private unnamed_addr constant [3 x i8] c"%s\00", align 1
@fmt_int_nnl = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@fmt_char_nnl = private unnamed_addr constant [3 x i8] c"%c\00", align 1
@fmt_float_nnl = private unnamed_addr constant [3 x i8] c"%g\00", align 1
@fmt_str_nl = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@empty_str = private unnamed_addr constant [1 x i8] zeroinitializer, align 1
@strptr = private unnamed_addr constant [6 x i8] c"hello\00", align 1
@strptr.1 = private unnamed_addr constant [7 x i8] c" world\00", align 1

; Function Attrs: nofree nosync nounwind readnone willreturn
declare void @llvm.donothing() #0

; Function Attrs: nofree nosync nounwind willreturn
declare i8* @llvm.stacksave() #1

; Function Attrs: nofree nosync nounwind willreturn
declare void @llvm.stackrestore(i8*) #1

declare i32 @printf(i8*, ...)

declare i32 @sprintf(i8*, i8*, ...)

declare void @memcpy(i8*, i8*, i64)

declare %struct.Vector* @Vector_new()

declare void @Vector_push(%struct.Vector*, i8**)

declare void @Vector_pop(%struct.Vector*)

```

```
declare i8* @Vector_get(%struct.Vector*, i32)

declare void @Vector_free(%struct.Vector*)

declare i8* @Str_new(i8*)

declare i8* @Str_concat(i8*, i8*)

declare void @Str_push(i8*, i8)

declare i1 @Str_compare(i8*, i8*)

declare i8* @Str_clone(i8*)

declare void @Rng_init(i32)

declare i32 @Rng_generate(i32, i32)

declare void @exit(i32)

define void @main() {
entry:
  %mallocd_string = call i8* @Str_new(i8* getelementptr inbounds ([6 x i8], [6 x i8]* @strptr, i32 0, i32 0))
  %h = alloca i8*, align 8
  store i8* %mallocd_string, i8** %h, align 8
  %mallocd_string1 = call i8* @Str_new(i8* getelementptr inbounds ([7 x i8], [7 x i8]* @strptr.1, i32 0, i32 0))
  %w = alloca i8*, align 8
  store i8* %mallocd_string1, i8** %w, align 8
  %h_loaded = load i8*, i8** %h, align 8
  %w_loaded = load i8*, i8** %w, align 8
  %w_loaded2 = load i8*, i8** %w, align 8
  %h_loaded3 = load i8*, i8** %h, align 8
  %concatted_string = call i8* @Str_concat(i8* %h_loaded3, i8* %w_loaded2)
  %concatted = alloca i8*, align 8
  store i8* %concatted_string, i8** %concatted, align 8
  %c_ref = alloca i8**, align 8
```

```
  store i8** %concatted, i8*** %c_ref, align 8
  %c_ref_loaded = load i8**, i8*** %c_ref, align 8
  %print_arg = load i8*, i8** %c_ref_loaded, align 8
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
   [3 x i8]* @fmt_str_nnl, i32 0, i32 0), i8* %print_arg)
  %printf4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
   [4 x i8]* @fmt_str_nl, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8],
   [1 x i8]* @empty_str, i32 0, i32 0))
  %c_ref_loaded5 = load i8**, i8*** %c_ref, align 8
  %loaded_str = load i8*, i8** %c_ref_loaded5, align 8
  call void @Str_push(i8* %loaded_str, i8 33)
  %c_ref_loaded6 = load i8**, i8*** %c_ref, align 8
  %print_arg7 = load i8*, i8** %c_ref_loaded6, align 8
  %printf8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
   [3 x i8]* @fmt_str_nnl, i32 0, i32 0), i8* %print_arg7)
  %printf9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
   [4 x i8]* @fmt_str_nl, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8],
   [1 x i8]* @empty_str, i32 0, i32 0))
  %str_malloc_to_free = load i8*, i8** %concatted, align 8
  tail call void @free(i8* %str_malloc_to_free)
  %str_malloc_to_free10 = load i8*, i8** %h, align 8
  tail call void @free(i8* %str_malloc_to_free10)
  %str_malloc_to_free11 = load i8*, i8** %w, align 8
  tail call void @free(i8* %str_malloc_to_free11)
  ret void
}

declare void @free(i8*)

attributes #0 = { nofree nosync nounwind readnone willreturn }
attributes #1 = { nofree nosync nounwind willreturn }
```

6.3.2 pos_tuple_indexing – testing the functionality of shallow/deep tuple indexing

Listing 4: Source

```
pipe main [] |> unit {
  mut [int * [int * int]] t = tuple(4, tuple(4, 5));

```

```
  &int deep = &t[1][1];
  &int shallow = &t[0];

  Printnl <| [deep];
  Printnl <| [shallow];

  |> ();
}
```

Listing 5: Output

```
5
4
```

Listing 6: Target

```
; ModuleID = 'Platypus'
source_filename = "Platypus"

%struct.Vector = type { i32, i32, i8* }

@fmt_str_nnl = private unnamed_addr constant [3 x i8] c"%s\00", align 1
@fmt_int_nnl = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@fmt_char_nnl = private unnamed_addr constant [3 x i8] c"%c\00", align 1
@fmt_float_nnl = private unnamed_addr constant [3 x i8] c"%g\00", align 1
@fmt_str_nl = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@empty_str = private unnamed_addr constant [1 x i8] zeroinitializer, align 1

; Function Attrs: nofree nosync nounwind readnone willreturn
declare void @llvm.donothing() #0

; Function Attrs: nofree nosync nounwind willreturn
declare i8* @llvm.stacksave() #1

; Function Attrs: nofree nosync nounwind willreturn
declare void @llvm.stackrestore(i8*) #1

declare i32 @printf(i8*, ...)

declare i32 @sprintf(i8*, i8*, ...)
```

```
25
26 declare void @memcpy(i8*, i8*, i64)
27
28 declare %struct.Vector* @Vector_new()
29
30 declare void @Vector_push(%struct.Vector*, i8**)
31
32 declare void @Vector_pop(%struct.Vector*)
33
34 declare i8* @Vector_get(%struct.Vector*, i32)
35
36 declare void @Vector_free(%struct.Vector*)
37
38 declare i8* @Str_new(i8*)
39
40 declare i8* @Str_concat(i8*, i8*)
41
42 declare void @Str_push(i8*, i8)
43
44 declare i1 @Str_compare(i8*, i8*)
45
46 declare i8* @Str_clone(i8*)
47
48 declare void @Rng_init(i32)
49
50 declare i32 @Rng_generate(i32, i32)
51
52 declare void @exit(i32)
53
54 define void @main() {
55 entry:
56   %tuple_ptr = alloca <{ i32, i32 }>, align 8
57   %tuple-gep.0 = getelementptr inbounds <{ i32, i32 }>, <{ i32, i32 }>* %
      tuple_ptr, i32 0, i32 0
58   store i32 4, i32* %tuple-gep.0, align 4
59   %tuple-gep.1 = getelementptr inbounds <{ i32, i32 }>, <{ i32, i32 }>* %
      tuple_ptr, i32 0, i32 1
60   store i32 5, i32* %tuple-gep.1, align 4
```

```
61  %tuple_ptr1 = alloca <{ i32, <{ i32, i32 }>* }>, align 8
62  %tuple-gep.02 = getelementptr inbounds <{ i32, <{ i32, i32 }>* }>, <{ i32, <{
      i32, i32 }>* }>* %tuple_ptr1, i32 0, i32 0
63  store i32 4, i32* %tuple-gep.02, align 4
64  %tuple-gep.13 = getelementptr inbounds <{ i32, <{ i32, i32 }>* }>, <{ i32, <{
      i32, i32 }>* }>* %tuple_ptr1, i32 0, i32 1
65  store <{ i32, i32 }>* %tuple_ptr, <{ i32, i32 }>** %tuple-gep.13, align 8
66  %t = alloca <{ i32, <{ i32, i32 }>* }>*, align 8
67  store <{ i32, <{ i32, i32 }>* }>* %tuple_ptr1, <{ i32, <{ i32, i32 }>* }>** %
     t, align 8
68  %instance_of_struct = load <{ i32, <{ i32, i32 }>* }>*, <{ i32, <{ i32, i32
     }>* }>** %t, align 8
69  %gep_on_instance = getelementptr inbounds <{ i32, <{ i32, i32 }>* }>, <{ i32,
      <{ i32, i32 }>* }>* %instance_of_struct, i32 0, i32 1
70  %instance_of_struct4 = load <{ i32, i32 }>*, <{ i32, i32 }>** %
     gep_on_instance, align 8
71  %gep_on_instance5 = getelementptr inbounds <{ i32, i32 }>, <{ i32, i32 }>* %
     instance_of_struct4, i32 0, i32 1
72  %deep = alloca i32*, align 8
73  store i32* %gep_on_instance5, i32** %deep, align 8
74  %instance_of_struct6 = load <{ i32, <{ i32, i32 }>* }>*, <{ i32, <{ i32, i32
     }>* }>** %t, align 8
75  %gep_on_instance7 = getelementptr inbounds <{ i32, <{ i32, i32 }>* }>, <{ i32
     , <{ i32, i32 }>* }>* %instance_of_struct6, i32 0, i32 0
76  %shallow = alloca i32*, align 8
77  store i32* %gep_on_instance7, i32** %shallow, align 8
78  %deep_loaded = load i32*, i32** %deep, align 8
79  %print_arg = load i32, i32* %deep_loaded, align 4
80  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
     [3 x i8]* @fmt_int_nnl, i32 0, i32 0), i32 %print_arg)
81  %printf8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
     [4 x i8]* @fmt_str_nl, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8],
     [1 x i8]* @empty_str, i32 0, i32 0))
82  %shallow_loaded = load i32*, i32** %shallow, align 8
83  %print_arg9 = load i32, i32* %shallow_loaded, align 4
84  %printf10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
      [3 x i8]* @fmt_int_nnl, i32 0, i32 0), i32 %print_arg9)
85  %printf11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
```

```
    [4 x i8]* @fmt_str_nl, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8],
    [1 x i8]* @empty_str, i32 0, i32 0))
  %instance_of_tuple = load <{ i32, <{ i32, i32 }>* }>*, <{ i32, <{ i32, i32
   }>* }>** %t, align 8
  %gep_on_instance12 = getelementptr inbounds <{ i32, <{ i32, i32 }>* }>, <{
   i32, <{ i32, i32 }>* }>* %instance_of_tuple, i32 0, i32 0
  %gep_on_instance13 = getelementptr inbounds <{ i32, <{ i32, i32 }>* }>, <{
   i32, <{ i32, i32 }>* }>* %instance_of_tuple, i32 0, i32 1
  %instance_of_tuple14 = load <{ i32, i32 }>*, <{ i32, i32 }>** %
   gep_on_instance13, align 8
  %gep_on_instance15 = getelementptr inbounds <{ i32, i32 }>, <{ i32, i32 }>* %
   instance_of_tuple14, i32 0, i32 0
  %gep_on_instance16 = getelementptr inbounds <{ i32, i32 }>, <{ i32, i32 }>* %
   instance_of_tuple14, i32 0, i32 1
  ret void
}

attributes #0 = { nofree nosync nounwind readnone willreturn }
attributes #1 = { nofree nosync nounwind willreturn }
```

6.3.3 pos_vec_clone – testing the functionality of vector deep-copy

Listing 7: Source

```
pipe main [] |> unit {
  /* depth = 1 */
  {
    mut vector[int] x = Vector_new <| [];
    Vector_push <| [~&x, 2048];

    mut vector[int] y = %x;

    {
        ~&int y_setter = Vector_get_mut <| [~&y, 0];
        @y_setter = 4028;
    }


    Printnl <| [Vector_get <| [&y, 0]];
```

```
    Printnl <| [Vector_get <| [&x, 0]];
  }

  /* depth > 1 */
  {
    mut vector[vector[int]] x = Vector_new <| [];
    mut vector[int] x_inner = Vector_new <| [];
    Vector_push <| [~&x_inner, 2048];
    Vector_push <| [~&x, x_inner];
    mut vector[vector[int]] y = %x;

    {
        ~&int y_setter = Vector_get_mut <| [Vector_get_mut <| [~&y, 0], 0];
        @y_setter = 4028;
    }

    Printnl <| [Vector_get <| [Vector_get <| [&y, 0], 0]];
    Printnl <| [Vector_get <| [Vector_get <| [&x, 0], 0]];
  }


  |> ();
}
```

Listing 8: Output

```
4028
2048
4028
2048
```

Listing 9: Target

```
; ModuleID = 'Platypus'
source_filename = "Platypus"

%struct.Vector = type { i32, i32, i8* }

@fmt_str_nnl = private unnamed_addr constant [3 x i8] c"%s\00", align 1
@fmt_int_nnl = private unnamed_addr constant [3 x i8] c"%d\00", align 1
```

```
@fmt_char_nnl = private unnamed_addr constant [3 x i8] c"%c\00", align 1
@fmt_float_nnl = private unnamed_addr constant [3 x i8] c"%g\00", align 1
@fmt_str_nl = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@empty_str = private unnamed_addr constant [1 x i8] zeroinitializer, align 1

; Function Attrs: nofree nosync nounwind readnone willreturn
declare void @llvm.donothing() #0

; Function Attrs: nofree nosync nounwind willreturn
declare i8* @llvm.stacksave() #1

; Function Attrs: nofree nosync nounwind willreturn
declare void @llvm.stackrestore(i8*) #1

declare i32 @printf(i8*, ...)

declare i32 @sprintf(i8*, i8*, ...)

declare void @memcpy(i8*, i8*, i64)

declare %struct.Vector* @Vector_new()

declare void @Vector_push(%struct.Vector*, i8**)

declare void @Vector_pop(%struct.Vector*)

declare i8* @Vector_get(%struct.Vector*, i32)

declare void @Vector_free(%struct.Vector*)

declare i8* @Str_new(i8*)

declare i8* @Str_concat(i8*, i8*)

declare void @Str_push(i8*, i8)

declare i1 @Str_compare(i8*, i8*)

```

```
declare i8* @Str_clone(i8*)

declare void @Rng_init(i32)

declare i32 @Rng_generate(i32, i32)

declare void @exit(i32)

define void @main() {
entry:
  %Vector_new = call %struct.Vector* @Vector_new()
  %x = alloca %struct.Vector*, align 8
  store %struct.Vector* %Vector_new, %struct.Vector** %x, align 8
  %malloccall = tail call i8* @malloc(i32 ptrtoint (i32* getelementptr (i32,
    i32* null, i32 1) to i32))
  %malloc_of_t = bitcast i8* %malloccall to i32*
  store i32 2048, i32* %malloc_of_t, align 4
  %malloc_casted_to_void = bitcast i32* %malloc_of_t to i8*
  %ref_of_malloc = alloca i8*, align 8
  store i8* %malloc_casted_to_void, i8** %ref_of_malloc, align 8
  %loaded_vec = load %struct.Vector*, %struct.Vector** %x, align 8
  call void @Vector_push(%struct.Vector* %loaded_vec, i8** %ref_of_malloc)
  %x_loaded = load %struct.Vector*, %struct.Vector** %x, align 8
  %Vector_new1 = call %struct.Vector* @Vector_new()
  %v_len_ptr = getelementptr inbounds %struct.Vector, %struct.Vector* %x_loaded
    , i32 0, i32 0
  %vector_length = load i32, i32* %v_len_ptr, align 4
  %cur_index = alloca i32, align 4
  store i32 0, i32* %cur_index, align 4
  br label %clone_while

clone_while:                                      ; preds = %clone_while_body,
   %entry
  %x4 = load i32, i32* %cur_index, align 4
  %clone_loop_cond = icmp slt i32 %x4, %vector_length
  br i1 %clone_loop_cond, label %clone_while_body, label %clone_merge

clone_while_body:                                 ; preds = %clone_while
```

```
  %stackptr_prebody = call i8* @llvm.stacksave()
  %i = load i32, i32* %cur_index, align 4
  %vector_item = call i8* @Vector_get(%struct.Vector* %x_loaded, i32 %i)
  %vector_item_casted = bitcast i8* %vector_item to i32*
  %malloccall2 = tail call i8* @malloc(i32 ptrtoint (i32* getelementptr (i32,
    i32* null, i32 1) to i32))
  %cloned_value = bitcast i8* %malloccall2 to i32*
  %loaded_value = load i32, i32* %vector_item_casted, align 4
  store i32 %loaded_value, i32* %cloned_value, align 4
  %ptr_of_cloned_child = alloca i8*, align 8
  %casted_to_push = bitcast i32* %cloned_value to i8*
  store i8* %casted_to_push, i8** %ptr_of_cloned_child, align 8
  call void @Vector_push(%struct.Vector* %Vector_new1, i8** %
    ptr_of_cloned_child)
  %x3 = load i32, i32* %cur_index, align 4
  %add = add i32 %x3, 1
  store i32 %add, i32* %cur_index, align 4
  call void @llvm.stackrestore(i8* %stackptr_prebody)
  br label %clone_while

clone_merge:                                      ; preds = %clone_while
  %y = alloca %struct.Vector*, align 8
  store %struct.Vector* %Vector_new1, %struct.Vector** %y, align 8
  %loaded_vec5 = load %struct.Vector*, %struct.Vector** %y, align 8
  %vector_item6 = call i8* @Vector_get(%struct.Vector* %loaded_vec5, i32 0)
  %vector_item_as_type = bitcast i8* %vector_item6 to i32*
  %y_setter = alloca i32*, align 8
  store i32* %vector_item_as_type, i32** %y_setter, align 8
  %derefed_mutborrow = load i32*, i32** %y_setter, align 8
  store i32 4028, i32* %derefed_mutborrow, align 4
  %loaded_vec7 = load %struct.Vector*, %struct.Vector** %y, align 8
  %vector_item8 = call i8* @Vector_get(%struct.Vector* %loaded_vec7, i32 0)
  %vector_item_as_type9 = bitcast i8* %vector_item8 to i32*
  %print_arg = load i32, i32* %vector_item_as_type9, align 4
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
    [3 x i8]* @fmt_int_nnl, i32 0, i32 0), i32 %print_arg)
  %printf10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
    [4 x i8]* @fmt_str_nl, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8],
```

```
    [1 x i8]* @empty_str, i32 0, i32 0))
  %loaded_vec11 = load %struct.Vector*, %struct.Vector** %x, align 8
  %vector_item12 = call i8* @Vector_get(%struct.Vector* %loaded_vec11, i32 0)
  %vector_item_as_type13 = bitcast i8* %vector_item12 to i32*
  %print_arg14 = load i32, i32* %vector_item_as_type13, align 4
  %printf15 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
    [3 x i8]* @fmt_int_nnl, i32 0, i32 0), i32 %print_arg14)
  %printf16 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
    [4 x i8]* @fmt_str_nl, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8],
    [1 x i8]* @empty_str, i32 0, i32 0))
  %v_struct = load %struct.Vector*, %struct.Vector** %y, align 8
  call void @Vector_free(%struct.Vector* %v_struct)
  %v_struct17 = load %struct.Vector*, %struct.Vector** %x, align 8
  call void @Vector_free(%struct.Vector* %v_struct17)
  %Vector_new18 = call %struct.Vector* @Vector_new()
  %x19 = alloca %struct.Vector*, align 8
  store %struct.Vector* %Vector_new18, %struct.Vector** %x19, align 8
  %Vector_new20 = call %struct.Vector* @Vector_new()
  %x_inner = alloca %struct.Vector*, align 8
  store %struct.Vector* %Vector_new20, %struct.Vector** %x_inner, align 8
  %malloccall21 = tail call i8* @malloc(i32 ptrtoint (i32* getelementptr (i32,
   i32* null, i32 1) to i32))
  %malloc_of_t22 = bitcast i8* %malloccall21 to i32*
  store i32 2048, i32* %malloc_of_t22, align 4
  %malloc_casted_to_void23 = bitcast i32* %malloc_of_t22 to i8*
  %ref_of_malloc24 = alloca i8*, align 8
  store i8* %malloc_casted_to_void23, i8** %ref_of_malloc24, align 8
  %loaded_vec25 = load %struct.Vector*, %struct.Vector** %x_inner, align 8
  call void @Vector_push(%struct.Vector* %loaded_vec25, i8** %ref_of_malloc24)
  %x_inner_loaded = load %struct.Vector*, %struct.Vector** %x_inner, align 8
  %malloccall26 = tail call i8* @malloc(i32 ptrtoint (%struct.Vector**
   getelementptr (%struct.Vector*, %struct.Vector** null, i32 1) to i32))
  %malloc_of_t27 = bitcast i8* %malloccall26 to %struct.Vector**
  store %struct.Vector* %x_inner_loaded, %struct.Vector** %malloc_of_t27, align
    8
  %malloc_casted_to_void28 = bitcast %struct.Vector** %malloc_of_t27 to i8*
  %ref_of_malloc29 = alloca i8*, align 8
  store i8* %malloc_casted_to_void28, i8** %ref_of_malloc29, align 8
```

```
146   %loaded_vec30 = load %struct.Vector*, %struct.Vector** %x19, align 8
147   call void @Vector_push(%struct.Vector* %loaded_vec30, i8** %ref_of_malloc29)
148   %x_loaded31 = load %struct.Vector*, %struct.Vector** %x19, align 8
149   %Vector_new32 = call %struct.Vector* @Vector_new()
150   %v_len_ptr33 = getelementptr inbounds %struct.Vector, %struct.Vector* %
        x_loaded31, i32 0, i32 0
151   %vector_length34 = load i32, i32* %v_len_ptr33, align 4
152   %cur_index35 = alloca i32, align 4
153   store i32 0, i32* %cur_index35, align 4
154   br label %clone_while36
155
156 clone_while36:                                    ; preds = %clone_merge63, %
        clone_merge
157   %x68 = load i32, i32* %cur_index35, align 4
158   %clone_loop_cond69 = icmp slt i32 %x68, %vector_length34
159   br i1 %clone_loop_cond69, label %clone_while_body37, label %clone_merge70
160
161 clone_while_body37:                               ; preds = %clone_while36
162   %stackptr_prebody38 = call i8* @llvm.stacksave()
163   %i39 = load i32, i32* %cur_index35, align 4
164   %vector_item40 = call i8* @Vector_get(%struct.Vector* %x_loaded31, i32 %i39)
165   %vector_item_casted41 = bitcast i8* %vector_item40 to %struct.Vector**
166   %malloccall42 = tail call i8* @malloc(i32 ptrtoint (%struct.Vector**
        getelementptr (%struct.Vector*, %struct.Vector** null, i32 1) to i32))
167   %cloned_value43 = bitcast i8* %malloccall42 to %struct.Vector**
168   %loaded_diaper = load %struct.Vector*, %struct.Vector** %vector_item_casted41
        , align 8
169   %Vector_new44 = call %struct.Vector* @Vector_new()
170   %v_len_ptr45 = getelementptr inbounds %struct.Vector, %struct.Vector* %
        loaded_diaper, i32 0, i32 0
171   %vector_length46 = load i32, i32* %v_len_ptr45, align 4
172   %cur_index47 = alloca i32, align 4
173   store i32 0, i32* %cur_index47, align 4
174   br label %clone_while48
175
176 clone_while48:                                    ; preds = %clone_while_body49
        , %clone_while_body37
177   %x61 = load i32, i32* %cur_index47, align 4
```

```
178   %clone_loop_cond62 = icmp slt i32 %x61, %vector_length46
179   br i1 %clone_loop_cond62, label %clone_while_body49, label %clone_merge63
180
181 clone_while_body49:                               ; preds = %clone_while48
182   %stackptr_prebody50 = call i8* @llvm.stacksave()
183   %i51 = load i32, i32* %cur_index47, align 4
184   %vector_item52 = call i8* @Vector_get(%struct.Vector* %loaded_diaper, i32 %
       i51)
185   %vector_item_casted53 = bitcast i8* %vector_item52 to i32*
186   %malloccall54 = tail call i8* @malloc(i32 ptrtoint (i32* getelementptr (i32,
       i32* null, i32 1) to i32))
187   %cloned_value55 = bitcast i8* %malloccall54 to i32*
188   %loaded_value56 = load i32, i32* %vector_item_casted53, align 4
189   store i32 %loaded_value56, i32* %cloned_value55, align 4
190   %ptr_of_cloned_child57 = alloca i8*, align 8
191   %casted_to_push58 = bitcast i32* %cloned_value55 to i8*
192   store i8* %casted_to_push58, i8** %ptr_of_cloned_child57, align 8
193   call void @Vector_push(%struct.Vector* %Vector_new44, i8** %
       ptr_of_cloned_child57)
194   %x59 = load i32, i32* %cur_index47, align 4
195   %add60 = add i32 %x59, 1
196   store i32 %add60, i32* %cur_index47, align 4
197   call void @llvm.stackrestore(i8* %stackptr_prebody50)
198   br label %clone_while48
199
200 clone_merge63:                                    ; preds = %clone_while48
201   store %struct.Vector* %Vector_new44, %struct.Vector** %cloned_value43, align
       8
202   %ptr_of_cloned_child64 = alloca i8*, align 8
203   %casted_to_push65 = bitcast %struct.Vector** %cloned_value43 to i8*
204   store i8* %casted_to_push65, i8** %ptr_of_cloned_child64, align 8
205   call void @Vector_push(%struct.Vector* %Vector_new32, i8** %
       ptr_of_cloned_child64)
206   %x66 = load i32, i32* %cur_index35, align 4
207   %add67 = add i32 %x66, 1
208   store i32 %add67, i32* %cur_index35, align 4
209   call void @llvm.stackrestore(i8* %stackptr_prebody38)
210   br label %clone_while36
```

```
211
212 clone_merge70:                                   ; preds = %clone_while36
213   %y71 = alloca %struct.Vector*, align 8
214   store %struct.Vector* %Vector_new32, %struct.Vector** %y71, align 8
215   %loaded_vec72 = load %struct.Vector*, %struct.Vector** %y71, align 8
216   %vector_item73 = call i8* @Vector_get(%struct.Vector* %loaded_vec72, i32 0)
217   %vector_item_as_type74 = bitcast i8* %vector_item73 to %struct.Vector**
218   %loaded_vec75 = load %struct.Vector*, %struct.Vector** %vector_item_as_type74
       , align 8
219   %vector_item76 = call i8* @Vector_get(%struct.Vector* %loaded_vec75, i32 0)
220   %vector_item_as_type77 = bitcast i8* %vector_item76 to i32*
221   %y_setter78 = alloca i32*, align 8
222   store i32* %vector_item_as_type77, i32** %y_setter78, align 8
223   %derefed_mutborrow79 = load i32*, i32** %y_setter78, align 8
224   store i32 4028, i32* %derefed_mutborrow79, align 4
225   %loaded_vec80 = load %struct.Vector*, %struct.Vector** %y71, align 8
226   %vector_item81 = call i8* @Vector_get(%struct.Vector* %loaded_vec80, i32 0)
227   %vector_item_as_type82 = bitcast i8* %vector_item81 to %struct.Vector**
228   %loaded_vec83 = load %struct.Vector*, %struct.Vector** %vector_item_as_type82
       , align 8
229   %vector_item84 = call i8* @Vector_get(%struct.Vector* %loaded_vec83, i32 0)
230   %vector_item_as_type85 = bitcast i8* %vector_item84 to i32*
231   %print_arg86 = load i32, i32* %vector_item_as_type85, align 4
232   %printf87 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
        [3 x i8]* @fmt_int_nnl, i32 0, i32 0), i32 %print_arg86)
233   %printf88 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
        [4 x i8]* @fmt_str_nl, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8],
        [1 x i8]* @empty_str, i32 0, i32 0))
234   %loaded_vec89 = load %struct.Vector*, %struct.Vector** %x19, align 8
235   %vector_item90 = call i8* @Vector_get(%struct.Vector* %loaded_vec89, i32 0)
236   %vector_item_as_type91 = bitcast i8* %vector_item90 to %struct.Vector**
237   %loaded_vec92 = load %struct.Vector*, %struct.Vector** %vector_item_as_type91
       , align 8
238   %vector_item93 = call i8* @Vector_get(%struct.Vector* %loaded_vec92, i32 0)
239   %vector_item_as_type94 = bitcast i8* %vector_item93 to i32*
240   %print_arg95 = load i32, i32* %vector_item_as_type94, align 4
241   %printf96 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8],
        [3 x i8]* @fmt_int_nnl, i32 0, i32 0), i32 %print_arg95)
```

```
242   %printf97 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
        [4 x i8]* @fmt_str_nl, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8],
        [1 x i8]* @empty_str, i32 0, i32 0))
243   %v_struct98 = load %struct.Vector*, %struct.Vector** %y71, align 8
244   %v_len = getelementptr inbounds %struct.Vector, %struct.Vector* %v_struct98,
       i32 0, i32 0
245   %stored_v_len = load i32, i32* %v_len, align 4
246   %cur_item = alloca i32, align 4
247   store i32 0, i32* %cur_item, align 4
248   br label %free_while
249
250 free_while:                                       ; preds = %free_while_body, %
      clone_merge70
251   %x106 = load i32, i32* %cur_item, align 4
252   %free_loop_cond = icmp slt i32 %x106, %stored_v_len
253   br i1 %free_loop_cond, label %free_while_body, label %free_merge
254
255 free_while_body:                                  ; preds = %free_while
256   %stackptr_prebody99 = call i8* @llvm.stacksave()
257   %x100 = load i32, i32* %cur_item, align 4
258   %vector_item101 = call i8* @Vector_get(%struct.Vector* %v_struct98, i32 %x100
       )
259   %vector_item_as_type102 = bitcast i8* %vector_item101 to %struct.Vector**
260   %v_struct103 = load %struct.Vector*, %struct.Vector** %vector_item_as_type102
       , align 8
261   call void @Vector_free(%struct.Vector* %v_struct103)
262   %x104 = load i32, i32* %cur_item, align 4
263   %add105 = add i32 %x104, 1
264   store i32 %add105, i32* %cur_item, align 4
265   call void @llvm.stackrestore(i8* %stackptr_prebody99)
266   br label %free_while
267
268 free_merge:                                       ; preds = %free_while
269   call void @Vector_free(%struct.Vector* %v_struct98)
270   %v_struct107 = load %struct.Vector*, %struct.Vector** %x19, align 8
271   %v_len108 = getelementptr inbounds %struct.Vector, %struct.Vector* %
       v_struct107, i32 0, i32 0
272   %stored_v_len109 = load i32, i32* %v_len108, align 4
```

```
  %cur_item110 = alloca i32, align 4
  store i32 0, i32* %cur_item110, align 4
  br label %free_while111

free_while111:                                    ; preds = %free_while_body112
    , %free_merge
  %x120 = load i32, i32* %cur_item110, align 4
  %free_loop_cond121 = icmp slt i32 %x120, %stored_v_len109
  br i1 %free_loop_cond121, label %free_while_body112, label %free_merge122

free_while_body112:                               ; preds = %free_while111
  %stackptr_prebody113 = call i8* @llvm.stacksave()
  %x114 = load i32, i32* %cur_item110, align 4
  %vector_item115 = call i8* @Vector_get(%struct.Vector* %v_struct107, i32 %
    x114)
  %vector_item_as_type116 = bitcast i8* %vector_item115 to %struct.Vector**
  %v_struct117 = load %struct.Vector*, %struct.Vector** %vector_item_as_type116
    , align 8
  call void @Vector_free(%struct.Vector* %v_struct117)
  %x118 = load i32, i32* %cur_item110, align 4
  %add119 = add i32 %x118, 1
  store i32 %add119, i32* %cur_item110, align 4
  call void @llvm.stackrestore(i8* %stackptr_prebody113)
  br label %free_while111

free_merge122:                                    ; preds = %free_while111
  call void @Vector_free(%struct.Vector* %v_struct107)
  ret void
}

declare noalias i8* @malloc(i32)

attributes #0 = { nofree nosync nounwind readnone willreturn }
attributes #1 = { nofree nosync nounwind willreturn }
```