The Iris Final Report

Valerie Zhang (valerie.zhang@tufts.edu)
Josh Kim (joshua.kim634361@tufts.edu)
Ayda Aricanli (ayda.aricanli@tufts.edu)
Timothy Valk (timothy.valk@tufts.edu)
Trevor Sullivan (trevor.sullivan@tufts.edu)

December 2023

Contents

1 Introduction

Iris is a general-purpose, object-oriented programming language that combines features from Java, C++, and Smalltalk. An Iris program is a series of classes – nearly everything is a class! As such, Iris supports inheritance and encapsulation.

Iris puts a spin on normal encapsulation conventions by introducing `permit` class members. These members are accessible by a class's sub-classes as well as the classes whose names are in its private collection `permitted`. However, unlike friend classes in C++, a class's permitted classes may only access the `permit` methods of the class, rather than all private methods.

2 Language Tutorial

2.1 Notation

Code is distinguished from prose by using this `font`. Grammar rules are distinguished using this *font*.

2.2 Welcome to Iris

Hey there! Welcome to Iris. Let's learn how to write, compile, and run our very first Iris program. Note that Iris programs are written in `.iris` files. Let's dive in.

2.3 Hello World!

As introduced above, an Iris program consists of a series of class definitions. The starting point for program execution, familiarly, is still a "main" function, but this "main" function must be defined in a definition of a "Main" class. As such, to write a simple "Hello World" Iris program, let's start with a "Main" class definition:

```
class Main () {
   ..  class definition in here
}
```

Within the "Main" class, a "main" function must be defined under the "public" label, with the "univ" keyword and a return type of int (more on both "public" and "univ" later):

```
class Main () {
   public:
   int univ main() {
      ..  function definition in here
   }
}
```

The Olympus class is a built-in class that contains standard input and standard output functions. Thus, "Hello world!" can printed by calling the `println` function of the Olympus class:

```
class Main () {
   public:
   int univ main() {
      Olympus.println("Hello world!");
   }
}
```

2.4 Compilation and Execution

You have now written a compilable "Hello world!" Iris program! To compile the program – let's call it "hello-world.iris" – simply type: `./irisc hello-world.iris`

To run the compiled program, simply type: `./hello-world.exe`

2.5 Expanding Our Program – Instantiation

With "Hello world!" under our belt, we can continue journeying through Iris with instantiation. Consider the following code:

```
class Goddess () {
   public:
      string name;
      void univ smite() {
         Olympus.print("Bam");
      }
   private:
      int followers;
}

class Main () {
   public:
      int univ main() {
         Olympus.print("Hello world!");
      }
}
```

We now have two classes – our "Main" definition from before, and "Goddess" which has "name" as a public member, "followers" as a private member, and "smite" as a public method. To call "smite", we must instantiate the Goddess class in the "main" function:

```
class Goddess () {
   public:
      string name;
      void univ smite() {
         Olympus.print("Bam!");
      }
   private:
      int followers;
}

class Main () {
   public:
      int univ main() {
         Olympus.print("Hello world!");
         Goddess my_goddess = new Goddess();
         my_animal.noise();
      }
}
```

We should now print "Hello World!" followed by "Bam!"

2.6 Expanding Our Program – Inheritance

A key feature of Iris is the ability for classes to inherit from other classes. A child class inherits all the members of its parent class. Let's define a Dog class and call "noise" from it in "main" in this manner:

```
class Goddess () {
   public:
      string name;
      void univ smite() {
         Olympus.print("Bam!");
      }
   private:
      int followers;
}

class Artemis of Goddess () {
   public:
      void univ motto() {
         Olympus.print("I am the goddess of wisdom.");
      }
   private:
      int arrows;
}

class Main () {
   public:
      int univ main() {
         Olympus.print("Hello world!");
         Artemis my_artemis = new Artemis();
         my_artemis.smite();
      }
}
```

Because Artemis inherits "smite" from Goddess, this should print "Hello World!" followed by "Bam!" once again!

2.7 Expanding Our Program – Permit

At this point, we've built a program that exploits the features of classes. Let's add an Iris-specific feature to top it off: `permit`. A special form of encapsulation, members under the `permit` label are only accessible to another class if they're in the permitted list of that class. Consider a new method, under `permit`, in the Goddess class:

```
class Goddess () {
   public:
      string name;
      void univ smite() {
         Olympus.print("Bam!");
      }
   permit:
      void univ cast_spell() {
         Olympus.print("A spell has been cast on the kingdom!");
      }
```

```
   private:
      int followers;
}
```

A class's permitted list is defined inside the parentheses at the top of the class. Currently, Goddess has no class names in its permitted list, so no class other than Goddess can access the "make_spell" function. For instance, the following code will result in a runtime error:

```
class Goddess () {
   public:
      string name;
      void univ smite() {
         Olympus.print("Bam!");
      }
   permit:
      void univ make_spell() {
         Olympus.print("A spell has been cast on the kingdom!");
      }
   private:
      int followers;
}

class Main () {
   public:
      int univ main() {
         Goddess my_goddess = new Goddess();
         my_goddess.make_spell();
      }
}
```

Main is not in the permitted list of Goddess, and thus, the above code fails. Let's introduce a "Kingdom" class and include it in Goddess's permitted list:

```
class Goddess (Kingdom) {
   public:
      string name;
      void univ smite() {
         Olympus.print("Bam!");
      }
   permit:
      void univ make_spell() {
         Olympus.print("A spell has been cast on the kingdom!");
      }
   private:
      int followers;
}

class Kingdom () {
   public:
      void univ cast_spell() {
         Goddess a_goddess = new Goddess();
         a_goddess.make_spell();
      }
}
```

```
class Main () {
   public:
      int univ main() {
         Kingdom my_kingdom = new Kingdom();
         my_kingdom.cast_spell();
      }
}
```

And voila! A spell has been cast on the kingdom! We are able to call "make_spell" in Kingdom because Kingdom is in the permitted list of Goddess. You now have the building blocks needed to write, compile, and execute interesting and complex Iris programs. Reference the following Language Manual for complete descriptions of syntax and semantics, and more code examples. Go forth!

3 Language Manual: Lexical Conventions

3.1 Comments

The characters `..` introduce a single-line comment.
The characters `.~*` introduce a multi-line comment, which terminates with `*~.`

```
..This is a single line comment
.~* This is a multi-
line comment *~.
```

3.2 Identifier Conventions

Any valid identifier in Iris is an uppercase or lowercase alphabetic character followed by any number of uppercase or lowercase alphabetic characters and/or digits. Uppercase and lowercase letters are distinct. The underscore `_` is also an alphabetic character, and an identifier starting with an underscore is valid.

3.3 Keywords

`if`	`of`	`bool`
`else`	`public`	`string`
`while`	`permit`	`float`
`return`	`private`	`true`
`class`	`void`	`false`
`self`	`univ`	`Olympus`
`new`	`int`	`Object`

3.4 Literals

Iris allows for literals for each primitive data type:

3.4.1 Integer literals

Integer → `0` | `-?[1 - 9]{1}[0 - 9]*`

An integer literal is a sequence of one or more digits. Integer literals can begin with exactly one "`-`", and if so, are assumed to be negative. The only integer literal that can start with 0 is 0. All integer literals are assumed to be in base 10.

3.4.2 Float literals

Float → `[` *Integer* `][.]{1}[0 - 9]+`

A floating literal consists of 3 parts: a sequence of one or more digits, the decimal point, and another sequence of one or more digits. All 3 of these parts MUST be present for a float literal. Floating literals can optionally begin with exactly one "`-`", and if so, are assumed to be negative. All float literals are assumed to be in base 10.

3.4.3 Boolean literals

Boolean → `true` | `false`

3.4.4 String literals

string → `["][` *ascii-literal* `]*["]`

A string literal consists of a series of character literals surrounded by double quotes. Within the string literal, a `"` character must be preceded by a single \.

4 Language Manual: Types

typ → *int* | *bool* | *float* | *string* | *void* | `Object of` *string*

Functions or methods (used interchangeably) can be declared to return and/or take in any of the above types as parameters. Variables can also be set as any of the types. When declaring a variable of some *Class* type (represented by `Object of` *string*) the name of the class encoded in the *string* is used rather than the word `Object`.

An `int` is represented by a 32-bit OCaml integer, where the leading 31 bits encode the integer itself, and the LSB is an integer indicator (if it's on, then the data represents an integer). A *float* is an OCaml float, and a *string* is represented as an OCaml string.

4.1 Primitive Types

Primitive types refer to every type except the `Object of` *string* type. Primitive types are distinguished by the fact that they are not classes. As such, there is no way of directly calling methods on these types with the `.` (dot) operator. Furthermore, creating new instances of these types is simpler than creating new instances of Object types.

```
..  The following line instantiates an integer
int number;
```

4.2 Object

The `Object of` *string* type refers to the pre-defined classes Object and Olympus, as well as any other user-defined classes. This type allows users to create and interact with instances of their own

user-defined classes and write methods returning user-defined classes. Unlike the primitive types, instances of Object variables must be initialized by using the class' corresponding constructor using the `new` keyword. See the Class Instantiation section for more information on `new`.

5 Language Manual: Expressions

expr	→	*literal*	
	|	*var*	5.1
	|	*uop expr*	5.2
	|	*expr postfix*	5.3
	|	*expr binop expr*	5.4
	|	*var assign expr*	5.5
	|	*typ id assign expr*	5.5
	|	*var opassign expr*	5.5
	|	*id.id(expr list)*	5.6
	|	`new` *id* `()`	5.7

5.1 Variable Access

var → *id* | *id.id*

There are 3 types of variables in Iris: formals (function parameters), locals, and class variables. Formals and locals are always accessed using their `id`. Class variables, on the other hand, are accessed differently depending on scope.

A class variable can be referred to with just its `id` if it is within the class it is defined in. Outside of that class, it is called with `id1.id2`, where `id1` is the identifier for an object instance of its class, and `id2` is the identifier for the class variable. If a child class accesses a variable it is overriding from its parent, it will access the most local definition of the variable defined at `id`.

5.2 Unary Operators

uop → `-` | `!`

5.2.1 `-`*expr*

In addition to being the binary operator for subtraction, `-` encodes the negation operator, a right-associative unary operator that negates its operand. It can only be used on an *expr* that evaluates to an `int` or a `float`.

5.2.2 `!`*expr*

The logical-not operator is a right-associative unary operator that is used to compare Booleans. Returns the Boolean literal `true` if its operand is `false`, and `false` if its operand is `true`. It can only be used on an *expr* that evaluates to type `bool`, or `int`. For `int`s, if the value evaluates to `0`, then it returns the Boolean literal `true`. Otherwise, it returns `false`.

5.3 Postfix Unary Operators

postfix → `++` | `--`

5.3.1 *id*`++`

The plus-plus operator is a left-associative unary operator and is used to increment integer variables by 1. Returns its operand, which must be a variable or class variable, plus one.

5.3.2 *id--*

The minus-minus operator is a left-associative unary operator and is used to decrement integer variables by 1. Returns its operand, which must be a variable or class variable, minus one.

5.4 Binary Operators

binop → `!=`
| `==`
| `+`
| `+=`
| `-`
| `-=`
| `*`
| `*=`
| `/`
| `/=`
| `>`
| `>=`
| `<`
| `<=`
| `||`
| `&&`

5.4.1 *expr* `!=` *expr*

The not-equals operator is a left-associative binary operator and is used to compare the equality of expressions that evaluate to the same primitive types. Returns Boolean `true` if its two operands are different. Does not work on strings.

5.4.2 *expr* `==` *expr*

The equal-equals operator is a left-associative binary operator and is used to compare the equality of expressions that evaluate to the same primitive types. Returns Boolean `true` if its two operands are the same. When used on string literals, this operator is syntactic sugar for a call to `Olympus.streq(...)`, which checks the structural equality of strings.

5.4.3 *expr* `+` *expr*

The plus operator is a left-associative binary operator and is used to add expressions that evaluate to a primitive type (`int` or `float`). Returns the sum of its two operands if they are integers or floats, and will concatenate its two operands if they are strings. If a float and an integer are added, the result will be a float.

5.4.4 *expr* `-` *expr*

The minus operator is a left-associative binary operator and is used to subtract expressions that evaluate to `float`s or `int`s. Returns the difference between its first operand and second operand. If one expression is a `float` and the other an `int`, then it will evaluate to a `float`.

5.4.5 *expr* `*` *expr*

The times operator is a left-associative binary operator and is used to multiply expressions that evaluate to `float`s or `int`s. Returns the product of its two operands. If one expression is a `float` and the other an `int`, then it will evaluate to a `float`.

5.4.6 *expr* / *expr*

The division operator is a left-associative binary operator and is used to divide expressions that evaluate to `float`s or `int`s. Returns the result of dividing its first operand by its second. If one expression is a `float` and the other an `int`, then it will evaluate to a `float`.

5.4.7 *expr* > *expr*

The greater-than operator is a left-associative binary operator that is used to compare expressions that evaluate to `int`s or `float`s. Returns Boolean `true` if its first operand is greater than its second operand and `false` otherwise. Each operand must evaluate to the same type.

5.4.8 *expr* >= *expr*

The greater-than-equals operator is a left-associative binary operator that is used to compare expressions that evaluate to `int`s or `float`s. Returns Boolean `true` if its first operand is greater than or equal to its second operand and `false` otherwise. Each operand must evaluate to the same type.

5.4.9 *expr* < *expr*

The less-than operator is a left-associative binary operator that is used to compare expressions that evaluate to `int`s or `float`s. Returns Boolean `true` if its first operand is less than its second operand and `false` otherwise. Each operand must evaluate to the same type.

5.4.10 *expr* <= *expr*

The less-than-equals operator is a left-associative binary operator that is used to expressions that evaluate to `int`s or `float`s. Returns Boolean `true` if its first operand is less than or equal to its second operand and `false` otherwise. Each operand must evaluate to the same type.

5.4.11 *expr* || *expr*

The or operator is a left-associative binary operator that is used on expressions that evaluate to `bool`s. Returns Boolean literal `true` if either of its operands evaluate to `true` and `false` otherwise.

5.4.12 *expr* && *expr*

The and operator is a left-associative binary operator that is used on expressions that evaluate to `bool`s. Returns Boolean literal `true` if both of its operands evaluate to `true` and `false` otherwise.

5.5 Assignment

These are the *binop* operations that also perform assignment. For all of the asignment operators, the right-hand side of the operator must evaluate to the same type as the left-hand side. The *opassign* assignment type can only be used on previously declared variables. All of these expressions return the expression on the right-hand side.

5.5.1 *var* = *expr*

The assignment operator is a left-associative binary operator and is used to assign variable values. A valid assignment expression consists of a right-hand side expression that evalutes to the same type as the variable.

5.5.2 *typ id = expr*

The assignment operator can also be used in conjuction with local variable declarations. A valid expression of this form consists of a right-hand side expression that evalutes to type typ.

opassign → `+=`
| `-=`
| `*=`
| `/=`

5.5.3 *var += expr*

The plus-equals operator is a left-associative binary operator and is used to add and assign expressions that evaluate to primitive types to variables. Returns the sum of its two operands if they are numeric and assigns the value to its left operand, which must be a previously declared variable. If its operands are of string type, performs concatenation similar to `+` and assigns.

5.5.4 *var -= expr*

The minus-equals operator is a left-associative binary operator and is used to subtract and assign expressions that evaluate to primitive types to variables. Returns the difference between its two operands and assigns the result to its left operand, which must be a previously declared variable.

5.5.5 *var *= expr*

The times-equals operator is a left-associative binary operator and is used to multiply and assign expressions that evaluate to primitive types to variables. Returns the product of its two operands and assigns the result to its left operand, which must be a previously declared variable.

5.5.6 *var /= expr*

The divide-equals operator is a left-associative binary operator and is used to divide and assign expressions that evaluate to primitive types to variables. Returns the quotient of its two operands same as divide, and assigns the value to its left operand, which must be a previously declared variable.

5.6 Function Calls

Function calls are made using the `id.id(expr list)` rule, where `expr list` is a series of comma-delimited expressions. There are two types of functions or methods that exist in Iris: class methods and instance methods.

For class methods, the first `id` would be the class name and the second *id* would be the class method name. This is followed by an `expr list` which may be empty or nonempty.

The same goes for instance methods where the first `id` would be the instance name and the second `id` would be the instance method name. This is followed by an `expr list` which may be empty or nonempty.

5.7 Object Instantiation

Object instantiation is done through class constructors. A class constructor is a method defined by the class name followed by parentheses `()`. For example, the `Dog` constructor would be `Dog()`.

To instantiate an object a class constructor much follow the `new` keyword: `new Dog();`. This will allocate a new instance of the particular class on the heap and return the address to it. This address

can be assigned to a declared object instance as seen below:

```
Dog d = new Dog();

Dog d2;
d2 = new Dog();
```

See section 8, Classes, (specifically 8.4).

6 Language Manual: Statements

stmt → {*stmt list*}
| *expr*;
| `return` *expr*;
| `if (` *expr* `) {` *stmt* `}`
| `if (` *expr* `) {` *stmt* `}` *else* `{` *stmt* `}`
| `while (` *expr* `) {` *stmt* `}`
| *typ id*;

6.1 Block

The block statement is a list of statements.

6.2 *expr*

The *expr* statement is one of the expressions defined in 4.

6.3 `return`

The *return* statement contains the keyword `return` followed by an expression. It is used to pass a value out of a function.

6.4 `if`

The `if` statement contains the keyword `if` followed by an expression evaluating to a boolean, a statement representing the true clause, and a statement representing the false clause.

6.5 `else`

The `else` statement can only be used after an `if`; if the boolean condition of the preceding `if` evaluates to `false`, then the statement following the `else` is evaluated.

6.6 `while`

The while statement contains the keyword `while` followed by an expression and a statement. It is used to execute the statement a certain number of times by iterating according to the expression.

6.7 Local Declarations

The local declaration statement is a *typ* followed by an *id*. It is used to create a local variable by binding a type to a name.

7 Language Manual: Declaration

declaration → *variable decl*
| *function decl*
| *class decl*

Declarations introduce entities into the program that include an identifier which can be used in the program to refer to it. There are three types of declarations: variable declarations, function declarations, and class declarations. Variables and functions can only be declared inside a class declaration. Each declaration is packaged with specifiers defined below.

7.1 Identifiers (Names)

The rules for identifiers are as follows:

identifier → `[A-Z | a-z]+[A-Z | a-z | 0-9 | _ ]*`

Each declaration type must have an identifier associated with it.

7.2 Specifiers

There are three types of specifiers in Iris and are defined below:

specifiers → *typ*
| *class-specifiers*
| *univ-specifier-opt*

7.2.1 Type specifiers

Type specifiers are required for variable/instance declarations, and function declarations. These specifiers are any type defined in Iris: a primitive data type or a defined-class in Iris which is referred to using an identifier.

7.2.2 Class specifiers

There are two class specifiers:

class-specifiers → `class`
| *of-identifier-opt*

of-identifier-opt → *nothing*
| `of` *identifier*

The keyword `class` is used to declare a class and is followed by an identifier.

The keyword `of` is used for inheritance to specify the parent class of a new class. This specifier is optional. If not included, the newly defined class will automatically inherit from the Object class. Below are examples of class declarations:

```
class Dog of Animal () { ... }   ..the class Dog inherits from class Animal
class Dog () { ... }             ..the class Dog inherits from class Object
```

7.2.3 The `univ` specifier

univ-specifier-opt → *nothing*
| `univ`

The keyword `univ` is an optional specifier used to declare a method as a class method, and is used within class declarations when declaring a classes' methods. Methods without the univ keyword are instance methods used on instances of a class. The univ keyword cannot be used on a classes member variables. Examples of this are below.

```
..this is a declaration with univ
int univ sum(int a, int b) {
   return a + b;
}

..this is a declaration without univ
int mult(int a, int b) {
   return a * b;
}
```

7.3 Variable Declarations

A variable declaration is defined as follows:

variable-decl → *typ identifier*
| *typ identifier* `=` *expr*

The first rule for variable declarations for class members which is similar to local variable declaration. See section 6, Statements, (specifically 6.8) for more information on local variable declarations.

The second rule allows for simultaneous variable declaration and assignment. See 4 for more on *expr*.

7.4 Method Declarations

A function or method declaration is defined as follows:

function decl → *typ univ-specifier-opt identifier* `(` *variable-decl-list* `){` *stmt* `}`

Like a variable declaration, a function declaration must start with a type specifier followed by an identifier. A function declaration also includes an optional list of arguments which is defined below:

variable-decl-opt → *nothing*
| *variable-decl-list*

A *variable-decl-list* is a list of variable declarations separated by a comma '`,`'. A function declaration will also contain a body of statements which is described in the following section. Below are examples of function declarations:

```
..this is a univ function that takes in 2 int arguments
int univ sum(int a, int b) {
     return a + b
}
```

```
..this is a function that returns the string "a"
string a() {
      return "a";
}
```

7.5 Class Declarations

A class declaration is defined as follows:

class-decl → `class` *identifier of-opt identifier* `(` *permit-classes-opt* `)` `{` *encap-opt-list* `}`

The *permit-classes-opt* is a list of classes that have access to a class' `permit` members and methods, which can be empty or nonempty. See 7.2.4 for more on `permit`. Members can be included in the `permitted` category even when the *permit-classes-opt* is empty. The parentheses `()` must be included in the declaration, even if there are no permitted classes. The *encap-opt-list* is a list of encapsulation states a class may contain. See 7.2 for more on encapsulation. The states may be listed in any order, and may or may not be included. As such, an empty class may be declared with no encapsulation states included. However, member variable and method declarations must be defined under an encapsulation state, else they are assumed to be `public`. The curly braces {} must be included in the declaration. Below are examples of class declarations:

```
..this is a declaration of a class Dog
class Dog of Animal () {
   public:
      int name;
      int age;
      string univ noise() {
         return "bark";
      }
}

..this is a declaration of a class Goddess
class Goddess () {
   public:
      void univ smite {
         ..  print using built-in Olympus class, see "Built-in Classes" for more
         Olympus.print("Bam");
      }
      string univ toString() {
         return "I am a goddess.";
      }
   permit:
      void grantWish(string wish) { }
   private:
      int followers;
}
```

8 Language Manual: Classes

Classes are declared using the `class` keyword followed by the name of the class. If the class inherits from a class other than `Object`, the name is followed by the keyword `of`, followed by another class name – the parent. The body of a class, containing member variables and functions, is surrounded by { and }. The order in which classes are defined matters in two cases:

1. Inheritance: Parent classes must be defined above child classes.
2. Member variables: A class can only have object member variables of classes that have been defined before it.

Outside of these cases, all objects may be instantiated freely within functions, regardless of order.

8.1 The Object Class

The Object Class is the root of all classes in that it is a superclass of every existing and user-defined class. If a class does not inherit from an already existing class, it inherently will inherit from Object.

8.2 Encapsulation

8.2.1 Overview

In Iris, classes have three encapsulation states represented by the keywords `public`, `permit`, and `private`. At least one of these keywords must be provided if a class has a nonempty body. If the class is empty, keywords can be omitted without compiler errors.

Within the body of a class, `public`, `permit`, and `private` are followed by a `:` character. The members and/or methods that follow the keyword – until either another keyword occurs or the end of the class is reached – have its encapsulation level.

8.2.2 `public`

Member variables and functions under the `public` keyword are accessible by any class.

8.2.3 `private`

Member variables and functions under the `private` keyword are accessible internally within that class. They cannot be called by a client of that class.

8.2.4 `permit`

Member variables and functions under the `permit` keyword are conditionally accessible by other classes based on the private collection of permitted class names, surrounded by parentheses, that follow the class name. If a class is included in the private collection, it gains access to the `permit` category as if it were public. Otherwise, `permit` members remain private.

The privately declared collection of zero or more class names is included after the class name declaration using the following syntax: `(ClassA, ClassB, ...)`. The set of parentheses can be empty but cannot be omitted. Two examples of this are:

```
 .~* Simple example given some class Dog *~.
class Dog () {
   ..  class definition in here
}

 .~* Cat inherits from Animal, and Dog has permission to Cat's permit members *~.
class Cat of Animal (Dog) {
   ..  class definition in here
}
```

8.3 Inheritance

Similarly to Java, child classes inherit member variables and functions from a parent class. Child classes can also override their parent's member variables and functions. A child class automatically gets access to the `permit` member variables of its parent, but parents do not automatically get access to that of their children (they must be explicitly added to the child's private collection of permitted classes for this). Children do not inherit their ancestors' permitted list, but they do have access to their ancestors' private and permitted members. See 8.2.4 for an example of a class declaration involving inheritance.

8.4 Instantiation

Classes are passed around as pointers under the hood in Iris, and must be instantiated with the `new` keyword as in the following example.

```
..this will allocate space on the heap for c
ClassName c = new ClassName();
```

Classes not instantiated with the `new` keyword are uninitialized and will result in undefined behavior if an access is attempted.

9 Language Manual: Built-In Classes

9.1 The Main Class

The `Main` class contains exactly one class method: `univ main()`, which runs any Iris program. A user must define this class for any driver program and include the code to run within it.

```
 .~* Example of Main given some defined class Dog *~.
class Main () {
   int univ main() {
      Dog newDog = new Dog();
      string out = newDog.toString();
      Olympus.print(out);
   }
}
```

9.2 The Olympus Class

The `Olympus` class is similar to the `System` class in Java. It contains helpful functions and fields for reading in standard input and printing to standard output or error. It cannot be instantiated again by a user, but user-defined classes may inherit from `Olympus` if they wish.

The Olympus class includes the following methods:

Return	Function	Description
`void`	`univ print(string out);`	Given a *string*, prints it to standard output.
`void`	`univ println(string out);`	Given a *string*, prints it to standard output with a newline terminating the string.
`void`	`univ printerr(string out);`	Given a *string*, prints it to standard error.
`string`	`univ readaline();`	Reads a single line from standard input and returns it as a `string`. A line is defined as a sequence of characters, terminated by a newline or EOF.
`bool`	`univ streq(string s1, string s2);`	Takes in two strings and checks their structural equality. The "==", when used on strings, is syntactic sugar for a call to this function.

Standard input, standard output, and standard error are accessed only through these functions. There is no notion of a "stream" in Iris that the user can declare and pass to other functions.

10 Language Manual: Scope

Similar to C, functions and variables are in scope after their declaration until the end of the block of code they appear in (blocks are denoted with { and }). The rest of this section describes the expected behavior for the scope of variables and functions in a variety of cases.

Formals and local variables declared in functions go out of scope when a function returns. Declaring two variables or two functions of the same name within the same scope is not allowed (e.g. two instances of the Dog class named `d` in `main()`, two member variables in the same class named `age`, or two functions declared in the `Dog` class as `void bark()`). Declaring two classes of the same name is also not allowed.

Declaring a local variable of the same name as a member variable is allowed, but the local variable overshadows the member variable for the scope of that function. A child member variable declaration overshadows a parent member variable of the same name.

Declaring a member function that is already included in a parent class definition is also allowed, and the child definition will override the parent definition.

A member variable and member function of the same name in the same scope is allowed (so `int age;` and `int age()` can both be members of a class).

11 Project Plan

11.1 Process

Our group decided early on that we wanted to pursue an object-oriented language. We were inspired by the approach of Smalltalk – where everything's an object – and combined it with the syntax and conventions of Java and C++. We added our own touch with the `permit` form of encapsulation and thus, Iris – who delivers messages in Greek mythology – was born.

Despite being a team of five, our team found it easy to coordinate meeting times throughout the semester. In general, all work was done during these meetings, with the occasional bug resolved or feature added individually, on one's own time. Since submitting "Hello World!", we had a roadmap of language features we wished to implement and systematically built up the language in this order.

11.2 Project Timeline

20 September, 2023: Project Proposal
16 October, 2023: Scanner and Parser
18 October, 2023: Language Reference Manual: First Draft
8 November, 2023: Hello World!
29 November, 2023: Extended Testsuite
15 December, 2023: Final Compiler

11.3 Roles and Responsibilities

As stated above, our group implemented the overwhelming majority of this project together. We took our individual roles with a grain of salt – while Valerie often took lead on testing, and Trevor often took the lead on system architecture, every member of our group had significant contributions to all parts of the project.

11.4 Development Environment

The Iris compiler is written in OCaml, a functional programming language. The final output of the compiler, a result of codegen, are LLVM instructions, generated by the OCaml LLVM API. The editor used was Visual Studio Code, and the LiveShare extension was often used for simultaneous collaboration. GitHub was used for version control.

11.5 Intermediary Versions

Version Overview

Our commit log in our repository was incorrect for some reason (totaled to about 20 commits even though we had upwards of 100). Don't trust the image; we worked pretty steadily through November.

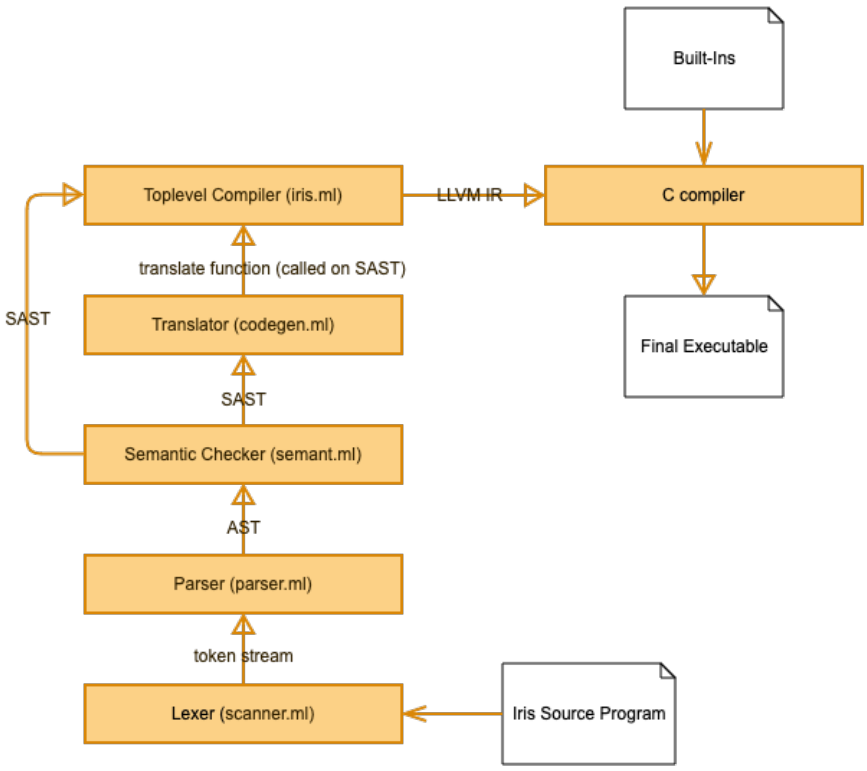
Here is the high-level prose overview of our versions.

1. Scanner/Parser with pretty-printed AST
2. Semant with pretty-printed SAST
3. Semant and Codegen: "Hello world!" and back-end setup for inheritance
4. Semant and Codegen: Incorporating much of Microc's lower-level code into class/program architecture (`exprs`, `stmts`, variables, etc.), testing via the `Main` class
5. Semant and Codegen: Implemented simple encapsulation with `public` and `private`
6. Semant and Codegen: Object instantiation, taking into account inheritance

7. Semant and Codegen: Implemented member functions with inheritance, including overloading and dynamic dispatch
8. Semant and Codegen: Implemented constructors for objects, including the dynamic dispatch case (e.g. `Animal a = new Cat();`)
9. Semant and Codegen: Implemented member variables: overloading, inheritance
10. Semant and Codegen: Implemented `univ` functions
11. Semant and Codegen: Implemented Iris `permit` encapsulation, including the lable and LLVM calls to a linked-in function that checks for permissions

12 Architectural Design

12.1 Architecture Diagram



12.1.1 Lexer (Whole Team)

The lexer, housed in `scanner.ml`, sweeps through an Iris program and compiles it into a series of tokens for use by the parser.

12.1.2 Parser (Whole Team)

The parser, housed in `parser.ml`, builds an abstract syntax tree of an Iris program from a series of tokens provided by the lexer.

12.1.3 Semantic Checker (Whole Team)

Housed in `semant.ml` and `gus.ml`, the semantic checker intakes an abstract syntax tree representing a program and, with the help of a large lookup table, outputs a semantically-checked abstract syntax tree of the program.

12.1.4 Translator (Whole Team)

Housed in `codegen.ml` and `guini.ml`, the translator intakes a semantically-checked abstract syntax tree of a program and emits an LLVM module representing the program. A lot of the work in this component was spearheaded by Trevor and Josh, although we all contributed to the writing, testing, and debugging of the generated LLVM.

12.1.5 Top-level Compiler (Ayda, Starter Code)

The Iris top-level compiler was mostly adapted from the MicroC top-level compiler. It coordinates the other steps of compilation and emits the final LLVM IR. Ayda wrote a lot of the LLVM code generation that linked Iris function calls to their respective C built-ins.

12.1.6 Built-Ins for `stdio` (Valerie, Ayda)

This is the small C file that links standard input and output functions and string conversions for primitive types, for use by our `Olympus` class' functionality. The file also contains a function for string comparison and a helpful function used in codegen for checking permitted classes, written by Trevor, Josh, and Tim. This is not available to user.

12.2 Implementation Summary

12.2.1 Feature: Classes and Inheritance

The whole team worked on implementing classes and inheritance. Since we needed a lookup table encoding all of the classes and the information contained within them for much of the semantic checking, we decided to take a first pass over the program and build a large lookup table that also encodes parent classes. In the semantic checker, we check class types and assignments by traversing through class' parents until we either hit Object, or the class we're looking for, to determine if it is a valid assignment. We also organize each class' functions to prepare to correctly instantiate their function virtual tables in the code generation phase; in order to handle overloaded functions in dynamic dispatch cases, the ordering of the virtual tables must match from parent to child class. Trevor and Valerie took the lead on implementing those features, but we all contributed to testing and debugging. We also implemented inheritance of member variables, which is something Josh took point on.

12.2.2 Feature: Encapsulation

The whole team worked on implementing `public` and `private` encapsulation levels in the semantic and code generation phases. We also all contributed to the semantic checking of permitted members, including the underlying design of the SAST. This included decisions about how to represent member variables and functions that fell under the `permit` label. Trevor, Josh, and Tim took the lead on implementing the code generation for (`permit`, which involved the runtime checking of permitted member variables and classes.

13 Test Plan

13.1 Testing Approach

We tested incrementally by hand as we added features, and often added these by-hand tests to our test suite. We have an automated test script that runs our test suite (the same as the ones submitted for intermediary deliverables) which we've run after every big change to ensure older cases still passed.

14 Lessons Learned

14.1 Valerie

These past few months have been such a whirlwind of a ride. I've had so much personal growth and growth in the relationship with my team. One lesson I learned is the importance of stepping up even if you don't feel confident in your knowledge or abilities. I had heard horror stories of groups with one slacking team member and I wanted to make sure I wasn't that type of teammate. However, I also had severe doubts in my ability to contribute to the project. That being said, I pushed myself to get into the weeds of codegen and semant while also keeping in mind the deliverables and deadlines we had coming up. Even when you don't think you can contribute to one portion, there is always something else that can be done whether that be going to office hours when we have a bug or working on other parts of a deliverable that are needed. One final thought and important realization that I've come to looking back on this project is that your team can really make or break the project. I was lucky to have found a team of all hardworking individuals that communicated well and I meshed with on a personal and professional level. I'm truly grateful for the opportunity to work with them and build something together.

To future students, do not underestimate codegen and please start early, I truly can't emphasize this enough. Even if you think implementing something will not be challenging, it definitely will be.

14.2 Ayda

Where to start? I feel like my brain has grown several sizes over the course of this semester, and yet I'm still learning something new every time we work on this compiler. I suppose the best thing I've learned is that who you work with on long-term projects really matters. I was really lucky to have a group this wonderful, but I can imagine how frightful it could be to have to work in a group that doesn't mesh well. The project has definitely made me gain a deep appreciation for how difficult building a compiler is. I was quite nervous coming into this class, as I've never been part of a long-term group project like this, and it covered things I've never touched before. I'm realizing how critical it is to have a solid understanding of both semantic theory and low-level implementation details for such a project. That was one of the challenges for me, since I find conceptual work easier and more enjoyable than the nitty-gritty of implementation. However, I feel like I've gained a new appreciation for the importance of how high-level design and low-level implementation work together, as opposed to thinking of each being important in its own vacuum, and learned how to think about both together. I'm really proud of the work we've done together on Iris, and I know I'll remember this course as one of my favorites (even if it thoroughly wiped the floor with me at times).

14.3 Tim

Compilers is unlike any class I've taken at Tufts. A semester-long group project where you're, in essence, thrown to the wolves is daunting, but it's been an incredible experience. I have learned so much about the interworkings of a compiler – how each phase is connected, how assembly gets generated, and the hurdles one has to clear to implement one. Codegen, especially, was a challenging

part of the project to me. I never felt completely comfortable with the LLVM API, and that made it challenging to implement the ideas we visualized in C. I thoroughly enjoyed the Scanner / Parser and Semant phases and got the hang of OCaml pretty quickly. But the best part of the project was hands down the team I got to work with and got to know closely over the few months. I'm grateful to have had such hardworking and passionate team members. We all motivate each other to push through times of seg faults and confusion. This class is so hands-off, and although it can be frustrating to not be directly given the keys to success, it makes you learn more by being independent. Looking back, Iris is a piece of work I can feel incredibly proud of and accomplished by.

14.4 Trevor

This semester has been a spectacular, illuminating experience. When we decided to implement an object-oriented language for our compiler, I had no idea what that would look like at a machine level (turns out it's fairly complicated). I've realized that there are so many abstractions in programming (and in life in general) that we take for granted.

I've also learned so much about system and protocol design. Good modularity is key to success, but implementing it begs the question: what information do the other modules need? We changed our S-AST at least 5 times, each time because we realized that codegen needed access to extra information or the information had to be presented differently. We were lucky that the changes weren't too difficult to make; these kinds of major changes are impossible on other systems. Thorough and open-ended protocol design is incredibly important and makes implementing new features much easier (and, in some cases, possible). You can never put too much thought into your system's design.

Above all else, I've discovered that the most important part of any major project is your team. During this project, I've had the pleasure of working with passionate, persevering, and kind people. I was so lucky to have a team that I got along with on a personal and professional level. When I look back on this class in the future, I know that the first thing I'll remember is my team.

14.5 Josh

When I first joined Compilers, I remember being pretty scared because I had no idea what making a compiler would even entail – I can say with certainty that I had no idea about what llvm was at the start of the semester. However, the way the course is structured made it easy to ease into the entire process. After the final weeks of finishing the project, I think that my understanding of how everything works and comes together has greatly improved.

In particular, when I first started working on the project, I was completely confused by codegen. Part of the reason why it was so difficult was because it seemed like such a huge mountain to scale. The most important part of finishing this project was finding out how we could break the project into manageable parts. Having to learn about everything on the spot was definitely scary, but what made it all great was having a team around me that I could rely on. In the last month, we've had many struggles with getting Iris to work. Yet, everyone in the group would always put in the effort to understand and resolve the problem. It always felt like everyone was invested in the project, which made it easier to work together. This class was a great experience, and I'm thankful that I had such a friendly and smart team for the entire semester.

14.6 General Advice

Our greatest advice is not to underestimate the amount of time code generation, and figuring out the LLVM OCaml API, is going to take. That took up the vast majority of the time we spent on this project, and it mainly is implemented in the final third of the course. Plan ahead for that and start as early as possible. Building reference code in LLVM from other languages that implement

your special features also helped us a lot.

For more implementation-specific advice, be sure to work in subgroups and delegate tasks to different people, even if you do work mostly in a group setting. Manage your versions by committing often, just in case. It never hurts to be able to roll back to a working version. Another strategy that saved us a lot of time was using distinct modules and helper functions for our lookup tables, since we had to change our lookup tables several times as we discovered new information we needed to implement the code generation phase. By implementing these in somewhat distinct modules and accessing values by helper functions, we saved a ton of time each time we had to make a change in the data structure.