Project 4: Semantic Analysis

Due Tuesday, Nov 20, 2018 at 8pm

Contents

Overview **2**

Distribution Code **4**

- Parser Files 5
- Driver Files 5
- Errors 5
- Types 5
- Functions 6
- Compiler Overview 6
- Compiler Core 7
  - AST Functions 7
  - Environments 7
  - Base and Start Nodes 8
  - Declarations 8
  - Printing Functions 8
- Statements 9
- Expressions 9
- Testing Framework 9
- Test Cases 10
- Inheritance and Polymorphism 10

Phase 1: Finding Declarations **10**

Phase 2: Resolving Types **11**

Phase 3: Resolving Function Calls **11**

Phase 4: Checking Fields and Variables **11**

Phase 5: Computing and Checking Types **11**

Phase 6: Checking Basic Control Flow **12**

Phase 7 (Optional): Advanced Control Flow **12**

Testing and Evaluation **12**

Grading **12**

Submission **13**

In this project, you will implement a semantic analyzer for uC, a small language in the C family. The main purpose of this project is to gain a deeper understanding of type systems and how a compiler works. Secondary goals are to write a substantial piece of object-oriented code in Python and to gain practice with inheritance, modularity, and code reuse.

As part of the code distribution for this project, we have provided you with a lexer and parser for uC, as well as a framework for a uC compiler. In this spec, we will provide suggestions for how to implement the remaining pieces of the semantic analyzer. You may make any modifications you like to the files marked with "modify it" in Distribution Code, as long as the interface remains the same.

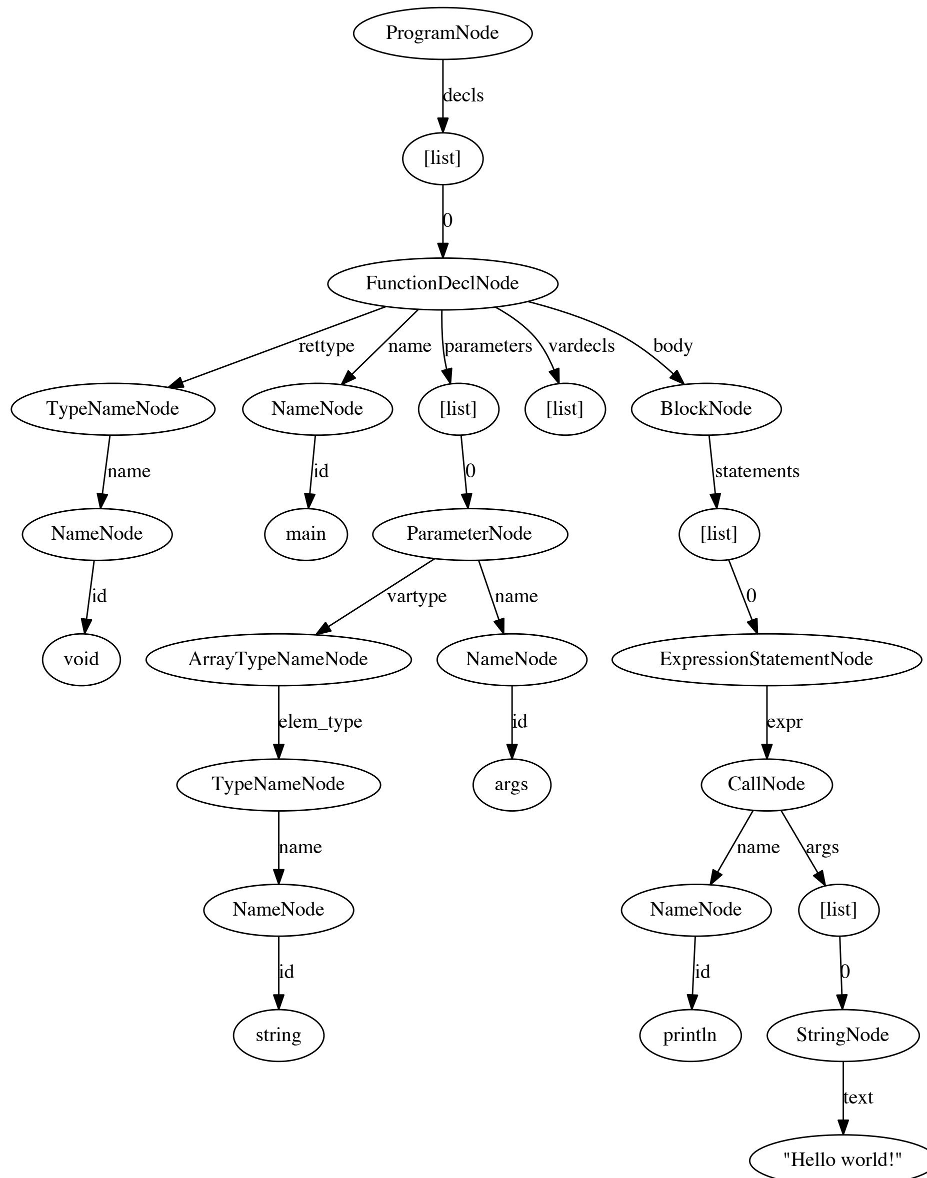
The project is divided into multiple phases, and we recommend completing them in the order specified.

Overview

A compiler generally consists of multiple phases of analysis and transformation over a source program. The first phase is usually parsing, which includes lexical analysis to divide the program into tokens and syntax analysis to organize the tokens into a format that represents the syntactic structure of the code. The parsing phase produces an *abstract syntax tree (AST)*, which is closely related to a derivation tree formed by deriving a program from a grammar. The main difference is that operations are generally encoded in the type of a node in an AST, rather than appearing on their own as a leaf node. As an example, consider the following uC program:

```
void main(string[] args)() {
  println("Hello world!");
}
```

The following AST is produced by the parser in the distribution code:



The nodes in the diagram above are either AST nodes (if the name ends with `Node`), lists, or *terminals*, which are tokens such as names, numbers, or strings. Each AST node has instance fields that refer to other nodes or terminals, represented by the edges in the picture above. The edge labels are the names of the corresponding fields. Edges from a list represent an element in the list, labeled by its index. The leaf nodes above are terminals, representing names such as `void` or `main`, or literals such as the string `"Hello world!"`.

Once a program has been transformed into an AST, the AST can be analyzed recursively in order to compute information, check correctness, perform optimizations, and generate code. The object-oriented paradigm is particularly well-suited to analyzing an AST, since each AST node class can define its own handling, with base functionality inherited from the base AST node class.

The phases in our compiler are the following, in order:

1. **Finding declarations.** The AST is traversed to collect all function and type declarations. In a uC program, a type or function can be used before its declaration.

2. **Resolving types.** Type names that appear in the AST are matched to the actual type that they name. An error is reported if a type name does not match a defined type.

3. **Resolving function calls.** Function calls that appear in the AST are matched to the function they name. An error is reported if a function name does not match a defined function.

4. **Checking fields and variables.** The fields in a type definition and the parameters and variables in a function definition are checked for uniqueness in each scope. Local environments are built for each scope, mapping each field, parameter, or variable name to its type.
5. **Computing and checking types.** Types are computed and stored for each expression in the program, and each use of a type is checked to ensure that it is valid in the context in which it is used.
6. **Checking basic control flow.** The AST is examined to check that all uses of the `break` and `continue` constructs occur in a valid context.
7. **Checking advanced control flow.** In this optional phase, local variables are checked to ensure that they are initialized before use, and non-`void` functions are checked to ensure that they can only exit through a `return` statement.

The compiler is run as follows:

```
$ python3 ucc.py -S <source file>
```

This parses the source file and then runs the semantic analysis phases described above. If any phase, including parsing, results in an error, the compiler exits after the phase completes, noting how many errors occurred. The `-S` argument restricts the compiler to semantic analysis, disabling the code generation phases that constitute the backend of the compiler.

If the `-T` command-line argument is present, then a representation of the AST with type information is saved to a file:

```
$ python3 ucc.py -S -T <source file>
```

The compiler implementation is divided into several Python files, described below.

Distribution Code

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs490.github.io/project-uc/frontend/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

Start by looking over the distribution code, which consists of the following files:

File or directory	Purpose	What you need to do with it
`lex.py`	Parser	Nothing
`yacc.py`	Parser	Nothing
`ucparser.py`	Parser	Nothing
`ucc.py`	Driver	Read it
`ucfrontend.py`	Driver	Read it
`ucbackend.py`	Driver	Nothing
`uccontext.py`	Contextual Information	Read and use it
`ucerror.py`	Errors	Read and use it
`ucbase.py`	Core	Read, use, and modify it
`uctypes.py`	Types	Read, use, and modify it
`ucfunctions.py`	Functions	Read, use, and modify it
`ucstmt.py`	Statements	Read, use, and modify it
`ucexpr.py`	Expressions	Read, use, and modify it
`uccheck.py`	Testing	Nothing
`Makefile`	Testing	Run it with `make`
`tests/`	Testing	Read, use, and modify it

Parser Files

The files lex.py and yacc.py are the source files for PLY, the parser generator we are using in this project. You do not have to be familiar with the code in these files.

The actual uC parser is specified in ucparser.py, in the format required by PLY. You do not have to examine the contents closely, but if you want to know more detail about how a particular AST node is constructed, the actual code to create a node accompanies the corresponding grammar rule in ucparser.py.

Running the parser generates a file called parsetab.py, which contains the internal data structures required by the parser. You do not have to be familiar with this file either, and we recommend excluding it from source control.

Driver Files

The top-level entry point of the compiler is ucc.py. It opens the given uC source file, invokes the parser to produce an AST, and then invokes each of the compiler phases on the AST. If the -T command-line argument is present, it then writes a representation of the AST, with annotated type information, to a file if no errors occurred in semantic analysis. If an error occurs during an analysis phase, the number of errors is reported and the compiler exits.

The PhaseContext class in uccontext.py records contextual information needed by a compiler phase. A new context can be created from an old one:

```
new_ctx = ctx.clone()
# modify new_ctx and pass it to child nodes
```

Contextual information is stored in the info dictionary of a PhaseContext as needed:

```
# whether or not the current node is within a loop
ctx.info['in_loop'] = False
```

The interface for the compiler phases is defined in ucfrontend.py. The top-level functions for each phase set the initial contextual information and call the appropriate method on the root AST node. In addition, the function for the type-checking phase also checks for a valid main() function, so you do not have to do so yourself.

Errors

The file ucerror.py defines the function you should use to report an error. The error() function takes as its first argument the current analysis phase, which you can extract from a PhaseContext. The function's second argument is the source position that resulted in the error. (Every AST node stores its position as an instance field, which can then be passed to error().) The third argument is the message to report to the user.

We evaluate correctness of your compiler by checking that the phases and line numbers in the errors you report match ours. The ucccheck.py file compares your output to the expected output included in the distribution files. See Testing and Evaluation for more detail on what constitutes correct output.

Types

The file uctypes.py contains definitions of classes that represent uC types. The Type class is the base class for all types, and it implements functionality common to all types. In particular, the array_type property is the array type for the given type. The lookup_field() method looks up a field in a type, and by default it produces an error. The mangle() method computes the name of the type to be used in code generation, which will not be used in this project.

The ArrayType class represents an array type, and it has a corresponding element type. The check_args() method checks that the given arguments are valid for constructing an array of the given type. You will need to complete the implementation of the check_args() and lookup_field() methods.

The PrimitiveType class represents a primitive type. You should not have to modify the class definition.

The UserType class represents a user-defined type. It is associated with the AST node that declares it, and it stores a reference to the set of fields defined by that node. You will need to complete the check_args() method, which determines whether or not the given arguments are valid for a constructor call for the given type. You will also need to fill in the definition for lookup_field().

You will not have to directly construct an object of Type or its derived classes. The distribution code will do so when a type is added to the global environment, as described below.

The is_compatible() function determines whether or not a source type matches or is convertible to a target type, as defined by the uC specification.

The is_numeric_type() function determines whether or not the given type is a numeric type.

The is_integral_type() function determines whether or not the given type is an integral type.

The `join_types()` function computes the type of a binary operation with the given operand types.

The `add_builtin_types()` function adds the built-in primitive types to the given dictionary. You will not need to call this function directly, as the distribution code does so when a global environment is created.

Functions

The file `ucfunctions.py` contains definitions of classes that represent uC functions. The `Function` class is the base class for all functions. A function has a name, a return type, and a list of of parameter types. The `mangle()` method computes the name of the function to be used in code generation. It will not be used in this project. The `check_args()` method checks whether the given arguments are compatible with the parameters of the function. You will have to fill in the definition.

The `PrimitiveFunction` class represents a built-in function. You will not need to modify this class.

The `UserFunction` class represents a user-defined function. It has an associated declaration node. When a `UserFunction` is created, the return type and parameter types are unknown. You will have to manually set the return type and call the `add_param_types()` method to record the parameter types when they become available in Phase 2.

You will not have to directly construct an object of `Function` or its derived classes. The distribution code will do so when a function is added to the global environment, as described below.

The `make_conversion()`, `add_conversion()`, and `add_builtin_functions()` functions construct the built-in uC functions and add them to the global environment. You will not have to call these functions, as the distribution code does so.

Compiler Overview

The following is a high-level overview of how the compiler works:

1. The parser reads a uC source file and produces an AST representation of the code. The root of the resulting AST is a `ProgramNode`, corresponding to the start variable of the grammar.

2. For each phase of the compiler, the top-level function in `ucc.py` sets up the context for that phase and then calls the `ASTNode` method for the phase on the root of the AST.

3. By default, the `ASTNode` method for a phase just recursively calls the method on the node's children. This method should be overridden where necessary. Due to dynamic binding, this results in customized computation when the recursion reaches a node with an override. The overriding method can recurse on the node's children with a super call, such as:

   ```
   def resolve_types(self, ctx):
       ... # customized computation
       super().resolve_types(ctx) # recurse on children
   ```

   The recursion terminates when it reaches terminals in the AST, which are not `ASTNode`s themselves.

In some cases, semantic analysis only requires information that is available directly in a node's children. For instance, the type of a binary expression such as `1 + 3.1` can be determined from the types of the operands, which are the `lhs` and `rhs` children of the binary operation. In other cases, however, the information is not directly available at a node's location. For example, the expression in a return statement must be checked against the return type of the function, the latter of which is not available directly to the return statement. Instead, such information is stored in the context that is the argument of an AST method. The contextual information is filled in where it is available, and then the context is passed down in recursive calls to the AST nodes that need that information. In the case of the return type, `ctx.info['rettype']` should be set in Phase 5 where the return type is known, so that the return statement can read it and check its expression against the expected type. You can see what contextual information is set for each phase in the top-level functions in `ucc.py`. The minimum information is the phase number and global environment.

As described in Driver Files, we recommend cloning a context before modifying it, so that the modification does not have the be undone when it is no longer relevant:

```
def basic_control(self, ctx):
    new_ctx = ctx.clone()
    new_ctx.info['in_loop'] = True # modify clone rather than original
    super().basic_control(new_ctx) # pass clone
    # original still has old value of in_loop
```

Compiler Core

The core of the compiler is defined in `ucbase.py`. This includes utility functions for operating on AST nodes, classes that represent environments, the base class for AST nodes, and nodes used in declarations.

AST Functions

The `astnode()` function produces a decorator that is used in defining an AST node class. An AST node has a position, which is the first argument that must be passed to its constructor. It also has *children*, which are AST nodes or terminals that must also be passed to the constructor. (A terminal is a token such as a name, integer, or string.) Finally, an AST node has *attributes*, which are not passed to the constructor. An attribute has a default value specified in the call to `astnode()`, and its value may be recomputed during semantic analysis.

A derived AST node class is defined by passing the children and attributes it introduces to `astnode()` and then applying the resulting decorator the the class definition for the node. A derived AST class inherits the children and attributes of its base class. AST classes that serve as "base nodes" do not define any children.

As an example, the following is the definition of `StructDeclNode`:

```
@astnode('name', 'vardecls', type=None)
class StructDeclNode(DeclNode):
    """An AST node representing a type declaration.

    name is the name of the type and vardecls is a list of field
    declarations. type is the instance of uctypes.Type that is
    associated with this declaration.
    """
    ...
```

The base class of `StructDeclNode` is `DeclNode`, which doesn't have any children or attributes itself. Thus, the only children of a `StructDeclNode` are those that it introduces, specifically `name` and `vardecls`. These are filled in by the parser when the `StructDeclNode` is created, and your code should not modify their values. The lone attribute of a `StructDeclNode` is `type`, since it is the only argument that is defaulted. It is initially `None`, but its value will be recomputed by your code during semantic analysis. Both children and attributes can be accessed as instance fields: if `node` is an instance of `StructDeclNode`, then `node.name`, `node.vardecls`, and `node.type` refer to the children and attributes.

The `ast_map()` function is used to map a function across the children of an AST node. A child may be a list, in which case the given function is mapped on the elements of the list. The optional `terminal_fn` argument, if given, is called on terminals that are not AST nodes. Refer to the methods of `ASTNode` for how `ast_map()` is used. You may not need to call `ast_map()` directly, since you can use `super()` to call the base class's implementation of a method.

Environments

The `GlobalEnv` class represents the global environment of a uC program. Only functions and types are contained in the global environment. Creating a `GlobalEnv` object automatically adds the built-in types and functions into the environment. You do not have to construct a `GlobalEnv`, since the provided driver does so.

The `add_type()` and `add_function()` methods take in a name and a definition node. They check whether or not an existing entity of the same category has the same name and produce an error if that is the case. Otherwise, they create an appropriate `Type` or `Function` object and add it to the global environment.

The `lookup_type()` and `lookup_function()` methods look up a type or function name in the global environment. If the `strict` argument is true and the name is not found, an error is reported, and a default type or function is returned. Otherwise, the type or function associated with the given name is returned. If the `strict` argument is false, no error is reported if the name is not found. You will only need to call these methods with the default `strict` as true. (The distribution calls them with `strict` as false when checking for a valid `main()` function.)

The `VarEnv` class represents a local scope, containing either the fields of a user-defined type or the parameters and variables of a user-defined function. The `add_variable()` method takes in a field, parameter, or variable name, its associated type, and a string representing the kind of entity being added. If another entity of the same name exists within the scope, an error is reported. Otherwise, the given name is mapped to the given type in the local scope.

The `contains()` method returns whether or not a name exists in the local scope. The `get_type()` method looks up a name and returns the associated type. If the name is not in scope, an error is reported, and a default type is returned.

You will need to create a `VarEnv` object for each user-defined type and function.

Base and Start Nodes

The `ASTNode` class is the base class for all AST nodes. It defines default implementations for each phase method. The `emit()` method is for code generation and will not be used in this project.

The `ProgramNode` class represents the start node, and it is the root of a program's AST. Its only child is a list of type and function declarations. You will not have to modify this class.

Declarations

The `DeclNode` class is the base class for type and function declarations. Like other classes that represent base nodes, it does not define any children.

The `StructDeclNode` class represents the declaration of a user-defined type. Its children are the name of the type, represented as a `NameNode`, and a list of field declarations, each represented as a `VarDeclNode`. The `type` attribute is the instance of `Type` associated with this declaration, which you should set in Phase 1. You will need to fill in the definition of `StructDeclNode`.

The `FunctionDeclNode` class represents the declaration of a user-defined function. Its children are the return type, represented as a `TypeNameNode` or `ArrayTypeNameNode`, the name of the function represented as a `NameNode`, a list of parameters each represented as a `ParameterNode`, a list of local-variable declarations each represented as a `VarDeclNode`, and a body represented as a `BlockNode`. The `func` attribute is the instance of `Function` associated with this declaration, which you should set in Phase 1. You will need to fill in the definition of `FunctionDeclNode`.

A `VarDeclNode` represents a variable or field declaration. It has children representing the type of the variable as a `TypeNameNode` or `ArrayTypeNameNode` and the name of the variable as a `NameNode`.

A `TypeNameNode` represents a simple type. It has a name, represented as a `NameNode`, and a `type` attribute that must be computed in Phase 2.

An `ArrayTypeNameNode` represents an array type. It has an element type, which is a `TypeNameNode` or `ArrayTypeNameNode`, and a `type` attribute that must be computed in Phase 2.

A `NameNode` represents a name in a uC program. It has an `id` child, which is a string containing the name.

A `ParameterNode` has the same structure as a `VarDeclNode`. It is included as a separate class for the purposes of Phase 7.

Printing Functions

The `child_str()` function is used in constructing a string representation of an AST. It is called by the `__str__()` method of an AST node. The resulting string representation, though it contains the full AST structure, does not present it in a very readable format, and it does not contain the attributes of an AST node. We recommend you use the `write_types()` method of an AST node to examine its representation instead, since it produces a more readable format and includes the `type` attribute for AST nodes that have the attribute.

Alternatively, you can call the `graph_gen()` function on a node, which will print a graph representation of the node and its children, in a format compatible with Graphviz, to the given output file. If you have Graphviz installed, you can then generate a PDF as follows:

```
$ dot -Tpdf -o <output file> <input file>
```

Alternatively, you can use Webgraphviz to generate an image from the output of `graph_gen()`.

Running `ucparser.py` directly on a source file will call `graph_gen()` on the resulting AST and write the result to an output file with extension `.dot`. The AST representation shown above is generated as follows:

```
$ python3 ucparser.py hello.uc
$ dot -Tpdf -o hello.pdf hello.dot
```

The `ucc.py` driver will also generate a `.dot` file, including type information for nodes that have been assigned a type, given the `-G` option:

```
$ python3 ucc.py -S -G hello.uc
$ dot -Tpdf -o hello.pdf hello.dot
```

The `.dot` file will only be generated if no errors are encountered. You can force it to be generated by disabling errors with the `-NE` option:

```
$ python3 ucc.py -S -NE -G hello.uc
$ dot -Tpdf -o hello.pdf hello.dot
```

You can also cutoff semantic analysis after a given phase to generate a graph after that phase:

```
$ python3 ucc.py -S --frontend-phase=2 -G hello.uc
$ dot -Tpdf -o hello.pdf hello.dot
```

We strongly recommend you make use of graph generation for debugging and testing.

Statements

The file `ucstmt.py` contains class definitions for AST nodes that represent statements, as well as the `BlockNode` class that represents a sequence of statements. Read through the distribution code for details about each class, and refer to the graph representation of a source file's AST if you are unsure about the structure of a node.

Expressions

The file `ucexpr.py` contains class definitions for AST nodes that represent expressions. The base class for all expressions is `ExpressionNode`, which has a `type` attribute that is computed in Phase 5. Like all attributes, this is inherited by derived classes.

The `LiteralNode` class is the base class for all literal expressions. It has a `text` attribute that consists of the literal text as used in code generation, which will not be used in this project. The derived classes define `text` as a child rather than an attribute, meaning that it must be passed to the constructor and appears when a node is printed.

The `NameExpressionNode` class represents a name used as an expression. Its child is a `NameNode` representing the actual name.

A `CallNode` represents a function call. Its children are the name of the function, represented as a `NameNode`, and a list of argument expressions. The `func` attribute refers to the instance of `Function` corresponding to the call, which you will compute in Phase 3.

A `NewNode` represents an allocation of an object of user-defined type. Its children are the name of the type, represented as a `NameNode`, and a list of argument expressions. You should compute the type of a `NewNode` expression in Phase 2.

A `NewArrayNode` represents an allocation of an array. Its children are the element type, represented as a `TypeNameNode` or `ArrayTypeNameNode`, and a list of argument expressions. You should compute the type of a `NewArrayNode` expression in Phase 2.

A `FieldAccessNode` represents a field access, and its children include an expression for the receiver object and the name of the field, represented as a `NameNode`.

An `ArrayIndexNode` represents an array-indexing operation, and its children include an expression for the receiver object and an index expression.

The remaining nodes in the file represent unary or binary operations. Pay close attention to the inheritance structure of the classes. You should attempt to write as much code as possible in base classes to avoid repetition in derived classes. Many of the derived classes should not require any code at all to be added to them.

A unary prefix operation consists of the expression on which the operation is to be applied and the name of the operation. A binary infix operation consists of the left-hand side expression, the right-hand side expression, and the name of the operation.

The `is_lvalue()` method of `ExpressionNode` determines whether or not the node produces an l-value. You will need to override this method where appropriate and make use of it where an l-value is required. It is defined as a static method in `ExpressionNode`, since it doesn't depend on the receiver object, but you may override it as either a static or instance method.

Testing Framework

A very basic testing framework is implemented in `uccheck.py` and the `Makefile`. The former takes in the name of a uC source file, such as `foo.uc`. It expects the result of running `ucc.py` on the source to be located in a file of the same name, but with `.uc` replaced by `.out`, as in `foo.out`. If compiling the source does not result in an error, it expects the type output to be in the file generated by `ucc.py`, e.g. `foo.types`. Finally, the correct outputs should be in files that end with `.correct`, as in `foo.out.correct` and `foo.types.correct`.

Running `make` will invoke `ucc.py` on each of the source files in the `tests` directory. It will then execute `ucccheck.py` to check the output against the correct files. Thus, you can use the `Makefile` rather than running `ucccheck.py` directly. You may need to change the value of the `PYTHON` variable in the `Makefile` if the Python executable is not in the path or not called `python3`.

You can also run an individual test through the `Makefile` by giving `make` a target that replaces the `.uc` extension with `.test`:

```
$ make tests/type_clash_user.test
```

Test Cases

The following basic test cases are provided in the `tests` directory, with compiler output. If a test has no errors, then the result of running the compiler with the `-T` command-line argument is also provided.

Correct Test	Description
`default.uc`	Test calling default constructors.
`equality.uc`	Test equality and inequality comparisons on primitive, user-defined, and array types.
`hello.uc`	Simple "hello world" example.
`length_field.uc`	Test accessing length field of arrays and structs.
`literals.uc`	Test literals of primitive type.
`particle.uc`	Complex example of a uC program.
`phase2.uc`	Test result of resolving types.
`use_before_decl.uc`	Test using types or functions before their declaration, which should be valid.

Error Test	Description
`bad_args.uc`	Test incorrect function and allocation arguments
`bad_array_index.uc`	Test incorrect array indexing
`bad_lvalue.uc`	Test using r-values in l-value contexts.
`bad_minus.uc`	Test the − operator with incorrect types.
`break_not_in_loop.uc`	Test `break` when it is not in a loop.
`function_clash_primitive.uc`	Test a user-defined function with the same name as a primitive
`type_clash_user.uc`	Test defining multiple types with the same name.
`unknown_function.uc`	Test calling an unknown function.
`unknown_type.uc`	Test using unknown types.
`variable_clash.uc`	Test repeated variable or parameter names.

These are only a limited set of test cases. You will need to write extensive test cases of your own to ensure your compiler is correct.

Inheritance and Polymorphism

There are a total of 62 node types that define a uC AST, arranged in a five-level class hierarchy rooted at `ASTNode`. You should take advantage of inheritance and polymorphism in order to customize the behavior of derived classes while avoiding repetition. In particular, you should place code that is common to sibling classes in their parent class, rather than repeating it. Also make use of `super()` calls when you need to add behavior to a derived class that is in addition to what the base class provides. With proper use of inheritance and polymorphism, you will find that you will need far fewer than the 372 (or 434 if you do Phase 7) methods you would if you repeat code for every phase in every class.

Phase 1: Finding Declarations

The first phase of semantic analysis is collecting type and function declarations, since types and functions can be used before their definition. For each type or function declaration, the type or function should be added to the global environment, referenced by `ctx.global_env`, using the appropriate insertion method. The latter will perform error checking for you to make sure a type or function is not redefined. Make sure to set the `type` and `func` attributes of a `StructDeclNode` and `FunctionDeclNode`, respectively.

When you are done with this phase, your compiler should pass the following tests:

```
$ make tests/type_clash_user.test tests/function_clash_primitive.test
```

You should also write test cases of your own, as the provided ones are not comprehensive.

Phase 2: Resolving Types

The second phase is to match occurrences of type names to the actual `Type` object named. In particular, you should compute the types of all `TypeNameNodes` and `ArrayTypeNameNodes`, as well as allocation nodes. Use `ctx.global_env` to look up a type name.

You will need to make use of recursion to resolve the types of `ArrayTypeNameNodes` and array-allocation nodes. For example, in order to compute the type of an `ArrayTypeNameNode`, its `elem_type` child must have its type computed first. You can then use the `array_type` property method of the resulting type to obtain the type of the `ArrayTypeNameNode`.

You will also need to use recursion to resolve types of subexpressions, such as in the following example:

```
new int[] { new int { 3 } }
```

Make use of `super()` to avoid repeating code.

You should also check in this phase that the `void` type is only used as a type name in the return type of a function. Set and use the contextual value `ctx.info['is_return']` to help check this.

Also make sure to record the return type and parameter types of a function in its associated `Function` object in this phase. Set the `rettype` attribute of a `Function` directly, and use the `add_param_types()` method to set the parameter types.

When you are done with this phase, your compiler should pass the following tests:

```
$ make tests/phase2.test tests/unknown_type.test
```

As always, you should write comprehensive tests of your own.

Phase 3: Resolving Function Calls

The third phase is to match function calls with the actual `Function` object that is being called. Use `ctx.global_env` to look up a function name.

When you are done with this phase, your compiler should pass the following tests:

```
$ make tests/unknown_function.test
```

Phase 4: Checking Fields and Variables

In the fourth phase, you should construct `VarEnv` objects representing the local scope of a user-defined type or function. This will ensure that field or parameter and variable names are distinct within a type or function, as well as record the type of each such entity. Store the `VarEnv` object for each type or function in an appropriate location, and then install the field or parameter and variable types in the `VarEnv`.

When you are done with this phase, you should pass the `variable_clash.uc` test.

Phase 5: Computing and Checking Types

This is the most substantial phase, as it involves computing the types of every expression and checking that a type is used in an appropriate context.

- You will need to set the contextual local environment and return type as appropriate in each function, so that you are able to look up variable names and check that a return statement is valid.
- For statements, you should check that their child expressions have the appropriate type according to the semantics of the statement and the current context.
- For each expression, you will need to compute its type according to its child expressions and terminals. Refer to the uC specification for what the type of an expression should be. Make use of the utility functions we provide you in `uctypes.py`.
- For simple allocations, ensure that the given type is not a primitive type (disallowing expressions such as `new int()`).
- For function calls and allocation expressions, you will need to check that the number of arguments and the argument types are compatible with the required types. Implement and make use of the `check_args()` methods on `Type` and `Function` objects. You will find the provided `is_compatible()` function useful here.

- For field accesses, implement and make use of the `lookup_field()` methods on `Type` objects. You should return the `int` type if the field is not defined and report an error.
- For array-indexing operations, you should assume that the result type is `int` if the receiver is not an array and report an error.
- In general, you should use the `int` type in the presence of an error if the type of an expression cannot be unambiguously determined. Use your best judgment on whether or not this is the case; it is not something we will test, except the specific cases mentioned above.
- Make sure to check for an l-value in contexts that require one. Implement and make use of the `is_lvalue()` method on `ExpressionNode` and its derived classes.

Refer to the uC specification for the type rules that must be met for each expression, which you are responsible for checking. Report an error if a rule is violated.

When you are done with this phase, you should pass all of the provided tests except `break_not_in_loop.uc`.

Phase 6: Checking Basic Control Flow

The final required phase is to check that `break` and `continue` statements occur within a loop. Use contextual information to determine whether or not this is the case.

When you are done with this phase, you should pass all of the provided test cases.

Phase 7 (Optional): Advanced Control Flow

In this phase, check control flow to make sure that a local variable is always initialized before use, and that a non-void function always returns a value. You do not have to treat a `while (true) { ... }` (or equivalent for loop) differently from any other loop.

In order to run this analysis phase, you will need to use the `--frontend-phase` command-line argument, which determines at which phase to end semantic analysis. The default is Phase 6, so the following overrides the default:

```
$ python3 ucc.py -S --frontend-phase=7 <source file>
```

This phase is optional and will not be graded. We will run your code with this phase turned off, with the default setting of ending at Phase 6.

Testing and Evaluation

We have provided a small number of test cases in the distribution code. However, the given test cases only cover a small number of possible errors, so you should write extensive tests of your own.

We will evaluate your compiler based on whether or not it reports an error on an erroneous input source file in the expected analysis phase on the expected source line numbers. You do not have to produce exactly the same number of errors as our solution, so long as the set of errors you report consist of the same phases and line numbers as ours. You also do not have to generate the exact error messages we do, but the error messages you use should be meaningful.

A compiler should never crash, even on erroneous input. Instead, your compiler should gracefully detect and report errors, and then avoid any actions that could result in a crash.

The code you write only needs to detect semantic errors, since the provided lexer and parser already detect grammatical errors. We will not test your code with input that is lexically or syntactically incorrect.

In addition to testing your error detection, we will also test your compiler to make sure it computes types correctly. We will compare the printed types of each AST node for a valid input source file with the expected types using the `-T` command-line argument to `ucc.py`.

Grading

The autograded portion of this project will constitute approximately 90% of your total score, and the remaining 10% or so will be from hand grading. The latter will evaluate the comprehensiveness of your test cases as well as your programming practices, such as avoiding unnecessary repetition and making appropriate use of inheritance and polymorphism. In order to be eligible for hand grading, your project must achieve at least half the points on the autograded portion (i.e. about 45% of the project total).

You are required to adhere to the coding practices in the course style guide. We will use the automated tools listed there to evaluate your code. You can run the style checks yourself as described in the guide.

Submission

All code that you write for the interpreter must be placed in `ucbase.py`, `ucexpr.py`, `ucfunctions.py`, `ucstmt.py`, or `uctypes.py`. We will test all five files together, so you are free to change interfaces that are internal to these files. You may not change any part of the interface that is used by `ucc.py` or `ucfrontend.py`.

Submit `ucbase.py`, `ucexpr.py`, `ucfunctions.py`, `ucstmt.py`, `uctypes.py`, and any of your own test files to the autograder before the deadline. We suggest including a `README.txt` describing how to run your test cases.