Project 4: Semantic Analysis

Contents

https://eecs390.github.io/project-uc/frontend/

1 Project 4: Semantic Analysis

1.1 *Optional checkpoint due Wednesday, Mar 27, 2024 at 8pm ET*

1.2 *Final deadline Wednesday, Apr 3, 2024 at 8pm ET*

2 Purpose

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 write a substantial piece of object-oriented code in Python and to gain practice with inheritance, modularity, and code reuse. You will also gain a basic understanding of how a compiler works.

2.1 Skills

This project will help you develop the following skills:

- navigating a large code base written by other people, identifying and understanding the pieces you need to know, and extending the code base with additional functionality
- working with a large class hierarchy and leveraging inheritance and polymorphism to avoid code duplication
- reading and analyzing a language specification to determine what makes code correct or incorrect
- synthesizing test cases that demonstrate whether your implementation is correct

2.2 Knowledge

The project will also help you become familiar with the following knowledge relevant to programming languages:

- abstract syntax trees and how they represent code structure
- the basics of program analysis and how compilers reason about code

3 Task

At a high level, your task in this project is to implement the semantic-analysis phases of a compiler. 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 the remainder of this spec, we will provide suggestions for how to implement the remaining pieces of the semantic analyzer. The project is divided into multiple phases, and you should complete an individual phase before moving on to the next one.

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes both code and test cases.

Start by reading the rest of this specification, which provides an overview of how the compiler works and the purpose of the individual files in the distribution code. The specification then guides you through each phase of the project. There is a lot of detail in this spec, and it's okay if not all of it is clear at first.

After taking a pass over the project spec, look over the uC language specification. You don't need to absorb everything there at once. Just try to get a big-picture idea of where things are located in that language spec, and you can come back to the details as you implement each phase of the project. You will find it useful to search through the project and language specs, using Ctrl-F or Cmd-F depending on your system, for relevant terms (e.g. searching for "l-value" in the language spec when implementing the `is_lvalue()` method).

Finally, we recommend doing a closer read over the following sections of the project and language specifications:

- The Program Structure section of the uC spec.
- The following sections of the project spec:
  - *Overview*
  - *Compiler Overview*
  - *Compiler Core*
  - *Inheritance and Polymorphism*

You can then get started on *Phase 1*.

Most of the code you write will actually be in *Phase 6*. The previous five phases only involve a small amount of code, and more of the effort will be in figuring out how things work.

We also recommend attending office hours and discussion to get up and running.

The following are the main subtasks for this project:

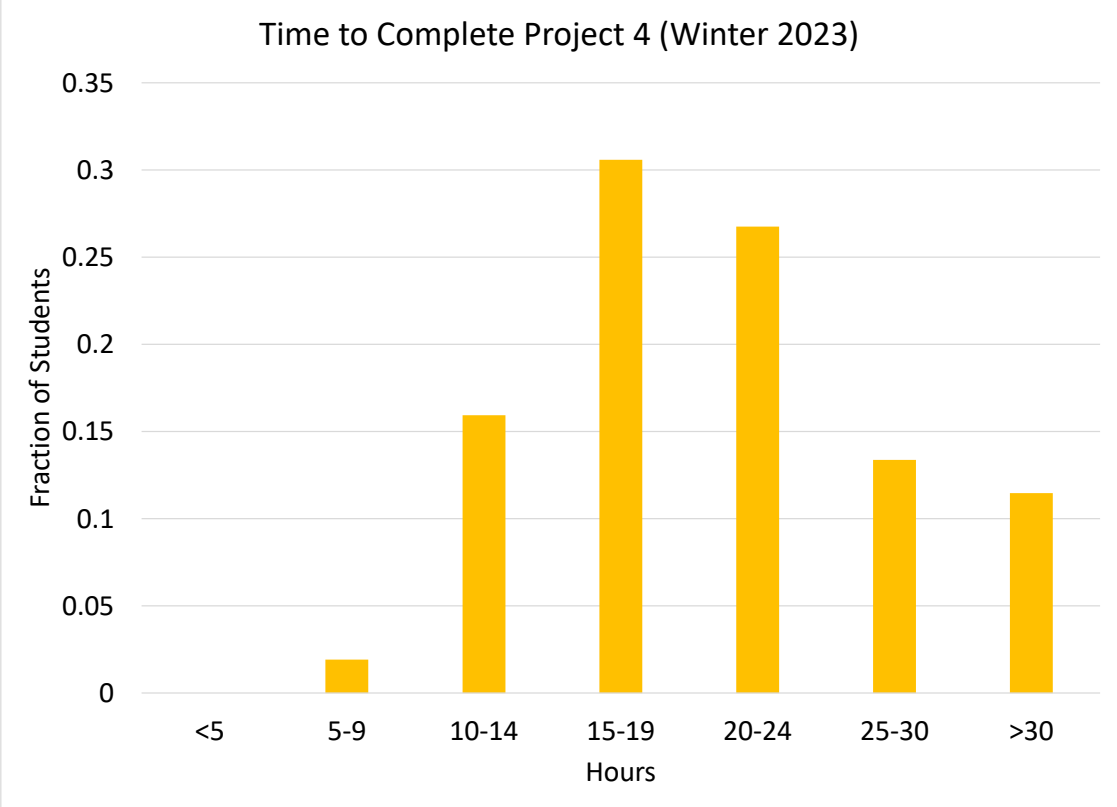
1. Complete the *Optional Checkpoint*, which is due on *Wednesday, Mar 27, 2024*. This requires earning 33 points on the autograder, including the automated style checks. With the public tests, you can achieve this by completing the first five phases, plus the beginning of *Phase 6* (through the `hello.uc` and `variables.uc` test cases).
2. Complete the rest of *Phase 6*. This will be the bulk of the code you write for the project. Once you pass all the public test cases excluding the mutation tests that grade your unit tests, you will have earned 62 out of the 107 points on the autograder (about 60%).

3. Take a closer look at each construct in the uC specification, paying close attention to what it mentions as requirements and what constitutes an error. Write test cases that have these errors, and ensure that your compiler passes these test cases. In addition, write some correct tests that exercise corner cases in the uC specification. Ensure that you get full credit on the mutation tests on the autograder; if your unit tests pass the mutation tests and your compiler passes your unit tests, this should also get you most of the private test cases on the autograder.
4. *Submit* your completed code by the final deadline, which is *Wednesday, Apr 3, 2024*.

Writing code that is readable, maintainable, and follows common practices is as important as writing code that is correct. Coding is a communal activity, and you will have colleagues on your future projects who will need to understand and work with your code. As such, make sure to adhere to the coding standards for this course, as well as object-oriented design principles.

3.1 Reported Time to Complete the Project

The following is the time students reported they spent on the project in Winter 2023.



These data are for planning purposes only. We do not consider the exact time spent on the project to be reflective of anyone's learning or ability. Rather, completing the project regardless of how much time it takes is what is important to achieve the learning goals of the project.

4 Criteria for Success

Successful completion of this project involves passing all the public test cases on the autograder, as well as a majority of the private cases. We do not expect you to pass every private test case – writing a bug-free compiler is an exceedingly difficult task, as evidenced by the extensive issue trackers for production compilers such as GCC and Clang.

In addition, a successful implementation must adhere to effective programming practices as mentioned above. We will hand-grade your code to evaluate your practices, looking for the following specific pitfalls:

- significant code duplication
- non-descriptive variable, function, or class names
- avoiding the recursive leap of faith, leading to unnecessary cases in recursive functions
- manually recursing on individual children where a `super()` call can be used instead (there are only a handful of instances where manually recursing is necessary)
- repeating work that has already been completed in a previous phase rather than just accessing or looking up the result
- use of `isinstance()`, `@singledispatch`, `match`, and similar patterns where method overriding can be used more cleanly (note that we do not prohibit all uses of `isinstance()`, `@singledispatch`, or `match`, just the ones that are unnecessary)
- uninformative documentation
- uninformative error messages
- modifications to the `ASTNode` class hierarchy
- unnecessary conditional or boolean logic (e.g. comparing to `True` or `False`, patterns such as "if test then true else false" instead of using the truth value of "test" directly, and so on)

Use the above as a checklist for making sure that your code meets the coding-practices requirements.

5 Optional Checkpoint

The checkpoint consists of achieving at least 30% of the points on the public and private test cases before the checkpoint deadline. Your grade for the checkpoint will be computed as the better of:

- $min(0.3, score)/0.3$, where $score$ is the fraction of autograded points earned by your best submission before the checkpoint deadline.
- $finalScore$, where $finalScore$ is the fraction of autograded points earned by your best submission before the final deadline.

Thus, completing the checkpoint is **optional**. However, doing it will work to your benefit, since you can guarantee full credit on the 20% of the project points dedicated to the checkpoint.

6 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 AST in Figure 1 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 and resolving name expressions.** The fields in a type definition and the parameters and variables in a function definition are checked for uniqueness. Local environments are built for each scope, mapping each field, parameter, or variable name to its type. Types are computed for each name expression.
5. **Checking basic control flow.** The AST is examined to check that all uses of the `break` and `continue` constructs occur in a valid context.
6. **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.
7. **Checking advanced control flow.** In this optional phase, 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:

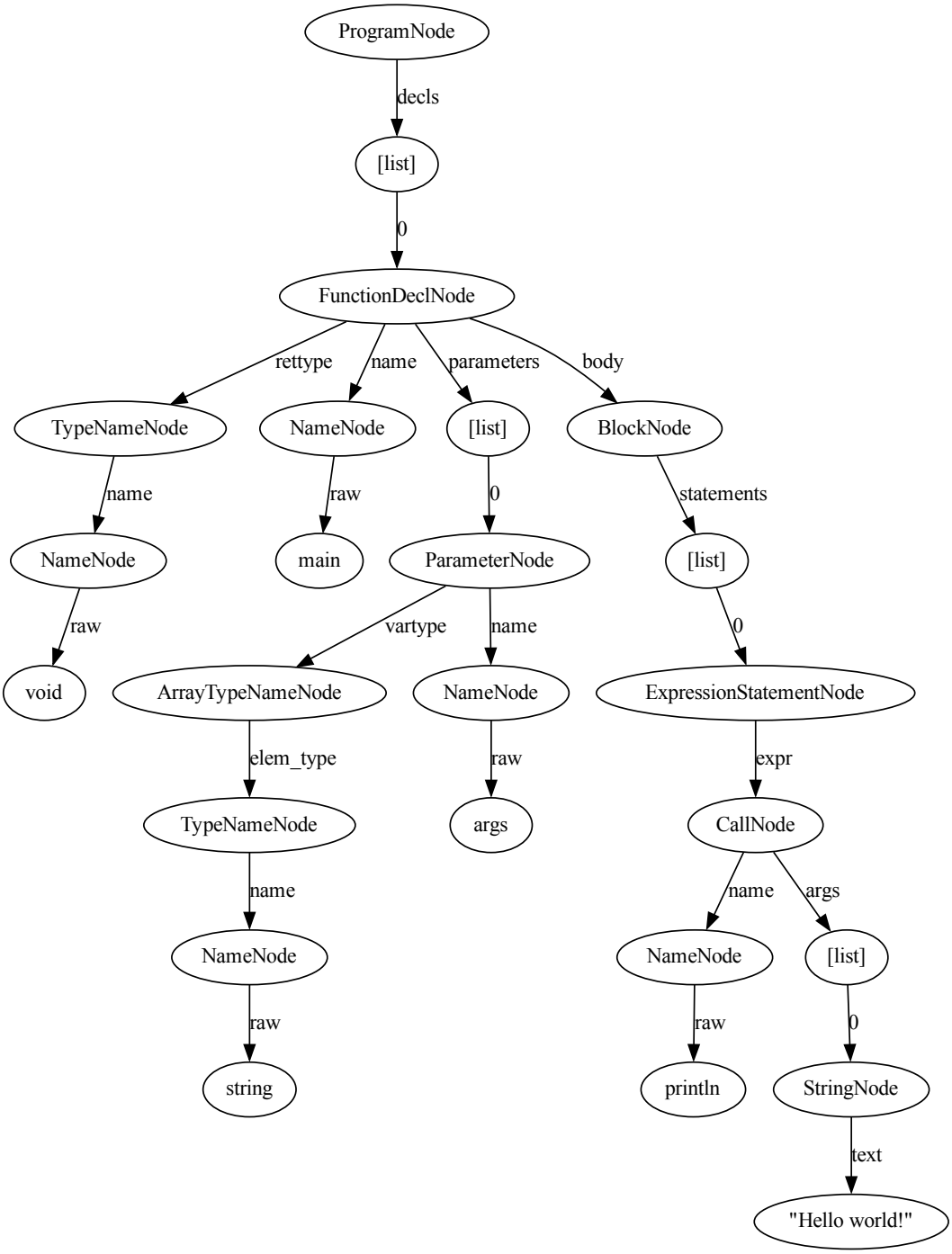


Figure 1: AST for a small uC program.

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

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

7 Distribution Code

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

```
$ wget https://eecs390.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
`ucccheck.py`	Testing	Nothing
`Makefile`	Testing	Run it with `make`
`tests/`	Testing	Read, use, and modify it

7.1 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`. However, you should **not** construct any AST nodes yourself - the parser will create the full AST for you.

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.

7.2 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 `PhaseContext`, which can be indexed as an associative container:

```
# whether or not the current node is within a loop
ctx['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.

7.3 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.

7.4 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 should not call this function directly, as the distribution code does so when a global environment is created.

7.5 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 should not call these functions, as the distribution code does so.

7.6 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 `ucfrontend.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['rettype']` should be set in *Phase 6* 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 `ucfrontend.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 to be undone when it is no longer relevant:

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

Alternatively, a context can be cloned and modified in one step:

```
def basic_control(self, ctx):
    super().basic_control(ctx.clone(in_loop=True))  # pass modified clone
    # original still has old value of in_loop
```

7.7 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.

Node Definitions

AST nodes are defined as Python 3.7 data classes ; we strongly recommend reading the documentation on data classes before proceeding with the project.

An AST node has the following data members:

- a unique ID (`node_id`), whose value is automatically generated when a node is created
- a position, which is the first argument that must be passed to its constructor
- zero or more *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)
- zero or more *attributes*, which are not passed to the constructor; instead, they are initially set to `None`, but their values may be recomputed during semantic analysis

The base `ASTNode` class defines the `node_id` and `position` members.

A node definition introduces children by defining class-level variables with a designated type but no initialization:

```
name: NameNode
fielddecls: List[FieldDeclNode]
```

An attribute is specified by defining a class-level variable with an `Optional` type and initializing it with a call to the `attribute()` function:

```
type: Optional[uctypes.Type] = attribute()
```

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`:

```
@dataclass
class StructDeclNode(DeclNode):
    """An AST node representing a type declaration.

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

    name: NameNode
    fielddecls: List[FieldDeclNode]
    type: Optional[uctypes.Type] = attribute()
    ...
```

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 `fielddecls`. 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 variable that is `Optional` and defaulted. It is initially `None`, but its value will be recomputed by your code during semantic analysis. Data members can be accessed as instance fields: if `node` is an instance of `StructDeclNode`, then `node.name`, `node.fielddecls`, and `node.type` refer to the children and attributes, and `node.position` refers to its position.

AST Functions

The `attribute()` function is a shorthand for constructing a data-class field that is defaulted to `None` and not passed to the constructor.

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. They return the resulting `Type` or `Function` object.

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 code 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. A `VarEnv` corresponding to a local scope within a function has a parent scope, while one corresponding to a user-defined type or function as a whole does not have a parent (the parent is `None`). The `add_variable()` method takes in a field, parameter, or variable name and its associated type. If another entity of the

same name exists within the scope itself or a parent 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 or its ancestors. 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, as well as for each construct that has its own local scope.

Base and Start Nodes

The `ASTNode` class is the base class for all AST nodes. It defines default implementations for each phase method that recursively call the method on each of the node's children.

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 `FieldDeclNode`. 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`, 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 `FieldDeclNode` represents a 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 a `raw` child, which is a string containing the name.

A `ParameterNode` has the same structure as a `FieldDeclNode`.

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 tests/hello.uc
$ dot -Tpdf -o tests/hello.pdf tests/hello.dot
```

We have also included a `Makefile` rule that invokes both the parser and Graphviz[1]:

```
$ make tests/hello.graph
Generating graph for tests/hello.uc...
python3 ucparser.py tests/hello.uc || true
Wrote graph to tests/hello.dot.
dot -Tpdf -o tests/hello.pdf tests/hello.dot
rm tests/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 tests/hello.uc
$ dot -Tpdf -o tests/hello.pdf tests/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 tests/hello.uc
$ dot -Tpdf -o tests/hello.pdf tests/hello.dot
```

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

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

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

7.8 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.

[1] Credit to Elliot Klein for this `Makefile` rule.

7.9 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 6*. Like all attributes, this is inherited by derived classes.

The `LiteralNode` class is the base class for all literal expressions. It has a `text` child that consists of the literal text as used in code generation, which will be implemented in a later project.

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 array or an object of user-defined type. Its children are the name of the type, represented as a `TypeNameNode` or `ArrayTypeNameNode`, and a list of argument expressions. You should compute the type of a `NewNode` 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.

7.10 Testing Framework

A very basic testing framework is implemented in `ucccheck.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
```

7.11 Test Cases

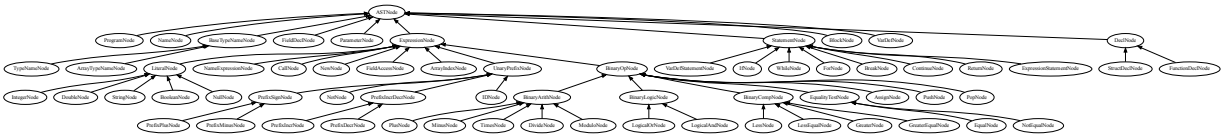
The following basic test cases are provided in the `tests` directory, with compiler output. If a test has no errors, then the result of running the compiler with the `-T` command-line argument is also provided.

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.
`variables.uc`	Test local variables.
`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
`self_init.uc`	Test a variable initialization referring to the variable itself.
`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.

7.12 Inheritance and Polymorphism



There are a total of 64 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 384 (or 448 if you do *Phase 7*) methods you would if you repeat code for every phase in every class.

8 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.

8.1 Internal Postconditions for Phase 1

Subsequent phases can assume the following postconditions hold after Phase 1:

- All `StructDeclNode`s have their `type` attribute filled in.
- All `FunctionDeclNode`s have their `func` attribute filled in.
- The global environment contains all user-defined types and functions (along with primitive types and functions, which are installed prior to Phase 1; note that array types are never contained in the global environment).

9 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 `TypeNameNode`s and `ArrayTypeNameNode`s, as well as allocation nodes (`NewNode`). Use `ctx.global_env` to look up a type name, and store the result in the `type` attribute of the AST node.

You will need to make use of recursion to resolve the type of an `ArrayTypeNameNode`. In particular, 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['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.

9.1 Internal Postconditions for Phase 2

Subsequent phases can assume the following postconditions hold after Phase 2, in addition to the postconditions for Phase 1:

- All descendants of `BaseTypeNameNode`s have their `type` attribute filled in.
- All `NewNode`s have their `type` attribute filled in.
- All `Function`s have their `param_type` and `rettype` fields filled in (this is already done for primitive functions prior to Phase 1).

10 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, and store the result in the `func` attribute of the AST node.

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

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

10.1 Internal Postconditions for Phase 3

Subsequent phases can assume the following postconditions hold after Phase 3, in addition to the postconditions for Phases 1 and 2:

- All `CallNode`s have their `func` attribute filled in.

11 Phase 4: Checking Fields and Variables and Resolving Names

In the fourth phase, you should construct `VarEnv` objects representing the scope of a user-defined type or function, and all local scopes contained within the latter. This will ensure that field or parameter and variable names are distinct within each scope, as well as record the type of each such entity.

You will need to store the `VarEnv` for a user-defined type in an appropriate location, as it will be needed in *Phase 6* to resolve the types of field accesses.

For local scopes within a function, use the contextual value `ctx['local_env']` to keep track of the current scope. You will need to use this as the parent scope when creating a new `VarEnv`, and you will need it for both variable definitions as well as resolving the type of name expressions.

Aside from a user-defined type or function itself, blocks (`BlockNode`) and for loops (`ForNode`) also have local scopes associated with them. Make sure to create a `VarEnv` with the appropriate parent and set the contextual `ctx['local_env']` value before recursing on their children.

A variable definition (`VarDefNode`) needs to add the variable to the current local environment. Also set the contextual `ctx['initializing']` value to the name of the variable, so that you can check whether a variable is used in its own initialization.

You will also need to check and resolve name expressions (`NameExpressionNode`) in this phase. Use the contextual local environment to look up and set the type of the node, and use the contextual `ctx['initializing']` value to check whether the name expression appears in the initialization of the variable itself, such as

```
int i = (i + 1); // error: initialization of i refers to i
```

When you are done with this phase, you should pass the `self_init.uc` and `variable_clash.uc` tests.

11.1 Internal Postconditions for Phase 4

Subsequent phases can assume the following postconditions hold after Phase 4, in addition to the postconditions for Phases 1 through 3:

- All user-defined types have an associated `VarEnv` that contains their fields, and the `VarEnv` is stored in an accessible location for each type (**not** in a context, since those do not persist between phases).
- All `NameExpressionNode`s have their `type` attribute filled in.

12 Phase 5: Checking Basic Control Flow

The fifth 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 the `break_not_in_loop.uc` test.

12.1 Internal Postconditions for Phase 5

Phase 5 does not have additional postconditions beyond what are guaranteed by Phases 1 through 4.

13 Phase 6: 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. Upon computing the type of an expression, store the result in the `type` attribute of the AST node for that expression.

Start by computing the type of literals. Refer to the uC specification for what the type of each literal is. You can use the global environment to look up a type by name.

Next, compute the type of a function call. Since uC does not have overloading, you do not need to check that the arguments are valid to compute the type of the call itself. You should pass the `hello.uc` and `variables.uc` test cases once you can compute the types of literals and calls.

Continue by implementing type checking on addition operations. You will find the `uctypes.join_types()` function useful for computing the resulting type. Refer to the uC specification for the requirements on the operands of the + operator. When you are done, you should pass the `literals.uc` test.

The remaining given test cases will require you to implement type computation and checking on substantially more expression and statement types. You should write your own test cases to allow you to implement and test incrementally.

- You will need to set the contextual return type as appropriate in each function, so that you are able to 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 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 user-defined types, you will need to make use of the saved `VarEnv` objects constructed in *Phase 4*.
- 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 test cases.

13.1 Internal Postconditions for Phase 6

Subsequent phases can assume the following postconditions hold after Phase 6, in addition to the postconditions for Phases 1 through 5:

- All descendants of `ExpressionNode` have their `type` attribute filled in.

14 Phase 7 (Optional): Advanced Control Flow

In this phase, check control flow to make sure 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.

15 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 consists 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 need to be meaningful – they should provide enough information for a user to diagnose the problem in their code.

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 compile-time, semantic errors. 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`.

16 Grading

The grade breakdown for this project is as follows:

- 20% checkpoint
- 70% final deadline autograded
- 10% final deadline hand graded

Hand grading will evaluate 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 solution must achieve at least half the points on the autograded, final-deadline portion of the project.

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.

16.1 Test Grading

We will autograde your uC tests for this project. Your test files must use the following naming pattern:

- `test-*.uc`

This pattern does not match any of the public test files – the buggy instructor solutions have been carefully crafted to pass all the public tests, so you will need to write your own tests that provide coverage distinct from the public tests.

You can submit up to 30 files matching the pattern above.

To grade your uC tests, we use a set of intentionally buggy instructor solutions (or *mutants*).

1. We run your uC tests with a correct solution, generating corresponding `.correct` files.
    - Tests that do not have syntax errors are valid.
    - Tests that have syntax errors are invalid.
2. We run all of your valid tests against each buggy solution, using the `.correct` files generated by the correct solution.
    - If any of your valid tests fail on the buggy solution, they have caught the bug.
    - The autograder assigns points for each bug that is caught.

You may include any number of errors and correct fragments in a single `test-*.uc` file. However, keep the following in mind:

- The compiler does not run later phases if an earlier phase produces an error. Therefore, do not include error cases that are supposed to be caught in different phases in the same test file – the cases from later phases will never be encountered.

- For checking correctness, we compare the set of (line number, phase) pairs where errors are reported. As such, try to avoid having multiple errors **on the same line**. For instance, the following can be problematic:

```
struct A {}; struct A {}; void print() {}
```

  The mutants are tested individually. If a mutant fails to detect duplicate types but does detect duplicate functions, it would report an error on this line (for the clash between the user-defined `print()` and the built-in one), so the set of (line number, phase) pairs would be the same as that for a correct implementation. As a result, this test case would fail to "kill" the mutant.

- The `.types` file is only generated and checked if a test case has no errors. Since this file is used to check whether an implementation correctly computes types, we recommend having separate files for error cases and for correct cases.

17 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 your own test files to the autograder before the deadline. You may submit up to 30 files for the pattern `test-*.uc` (do not submit `*.correct` files). If you have more test files than the limit, you will need to choose a subset to submit to the autograder.

18 Frequently Asked Questions

- **Q: How do I get started on the project?**

  A: First, if you haven't already, please follow the advice in the *Task* section of this project spec.

  Next, we recommend looking over `ucfrontend.py`. This has the interface functions for each phase of the compiler. They take the root node of the AST (a `ProgramNode`) as an argument, as well as a global environment (a `GlobalEnv`). Aside from setting up the context, they just invoke the corresponding method on the AST root.

  Then take a look at `ucbase.py`, where you'll find the definitions for `GlobalEnv` as well as `ASTNode`, which is the base class of all AST nodes (including `ProgramNode`). You'll see that it has a method for each phase, and the base behavior that it provides is to just recursively invoke the same method on each of its children. Thus, by default, what happens is that when the method is called on a `ProgramNode`, it calls the method on each of its children (which happen to be of type `StructDeclNode` or `FunctionDeclNode`), which call the method on their children, and so on until we reach the leaves of the AST.

  What you'll need to do is override the method for a phase in the necessary AST nodes so that when the recursion reaches them, you can do what is necessary to accomplish the work for that phase. For *Phase 1*, the method is `find_decls()`, and you'll need to override it in `StructDeclNode` and `FunctionDeclNode`. In most cases, when you override a method, you'll want to use a super call (e.g. `super().find_decls(ctx)`) to do the base-class work, e.g. invoking the method on all children. (This actually happens to be unnecessary in Phase 1, since uC does not have nested struct or function declarations.) You can then add the custom work for the derived class. You'll find that Phase 1 involves little work – you just need to obtain the global environment out of the context, call `add_type()` or `add_function()` with the appropriate arguments, and set the result to the `type` or `func` attribute of `StructDeclNode` or `FunctionDeclNode`, respectively.

  Later phases will require more work; the bulk of the work is in *Phase 6*. The earlier phases only require you to override the respective method in a handful of places. You'll need to figure out what the right places are. (For instance, when the project spec says "function calls", you'll need to determine what the right node type is. First,

observe that a function call is an expression, so the type is likely to be defined in `ucexpr.py`. Then look through `ucexpr.py` to find the appropriate class. It also helps to look through the uC language spec – the class names are similar to the names used for nonterminals in the grammar.)

- **Q: I get the error "Error (6) at line 1: signature for main must be void main(string[])" in phase 2. What is wrong?**

  A: This is usually the result of not doing the following from *Phase 2* correctly:

  > 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.

  Make sure that you are setting these to be `uctypes.Type` objects and not AST nodes.

- **Q: The type for node X is `None`. What could cause this?**

  A: First, make sure that you have overridden the appropriate method for computing the type of node X, either in X directly or one of its base classes. If you have done so, check whether or not the method ever gets invoked – you can do so either by adding a print statement, or by setting a breakpoint in your debugger. If the method never gets called, that means one of the parent nodes in the AST is failing to recurse on its children. Take a look at the AST structure for the test case, such as by using the graph generation discussed in *Printing Functions*. Find the instance of node X that is not having its method invoked and trace through its ancestor nodes in the AST. Do their methods get invoked? If you find a parent node whose method does get invoked, and its child's does not, then that parent node is likely missing a recursive call on the child (or a `super()` call to perform the recursion).

- **Q: Does the order of error messages matter?**

  A: No, as long as they are generated in the right phase. From *Testing and Evaluation*:

  > 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 consists of the same phases and line numbers as ours.

- **Q: What should I use as the `token` argument for creating a `uctypes.Type` or `ucfunctions.Function` object?**

  A: You should never manually create an object of `uctypes.Type`, `ucfunctions.Function`, or any of their derived classes. (The token is there specifically to prevent you from doing so.) Instead, add or look up the type or function in the global environment.

- **Q: Do we need to check for syntax errors in this project?**

  A: No. The parser is provided for you in this project, and it will detect all syntax errors.

- **Q: Do we need to check for runtime errors in this project?**

  A: No. We are doing compile-time analysis, so we are only checking for compile-time errors.