Project 5: Code Generation

Due Tue Dec 12 at 8pm

Contents

In this project, you will implement a code generator for uC, a small language in the C family. The main purpose of this project is to gain an understanding of the process of code generation and how one language can be translated into another. Secondary goals are to complete a compiler for a non-trivial language and to get practice with using macros and templates.

This project relies on the the first two phases of the semantic analyzer you implemented for uC in Project 4. We will only be testing this project with uC code that is semantically correct, so the later semantic analysis phases will be disabled. You will need to start with your Project 4 implementation as a basis, and we will only be providing distribution code for files that are different from or in addition to Project 4.

The project is divided into multiple phases that are described below.

Overview

In Project 4, you implemented a semantic analyzer for uC, which performed multiple analysis phases over a source program. Your main task in Project 5 is to implement the final code-generation phase, which will generate C++ code from an abstract syntax tree (AST). Since the compiler is generating C++ code rather than machine code, it is a *source-to-source* compiler.

As an example, consider the following uC program:

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

This can be translated into the following C++ code (which, modulo some minor spacing, is the actual code produced by the staff solution to this project):

```
#include "defs.h"
#include "ref.h"
#include "array.h"
#include "library.h"
#include "expr.h"

namespace _uc {

  // Forward type declarations

  // Forward function declarations

  _UC_PRIMITIVE(void)
     _UC_FUNCTION(main)(_UC_ARRAY(_UC_PRIMITIVE(string)) _UC_VAR(args));

  // Full type definitions

  // Full function definitions

  _UC_PRIMITIVE(void)
     _UC_FUNCTION(main)(_UC_ARRAY(_UC_PRIMITIVE(string)) _UC_VAR(args)) {
    {
      _UC_FUNCTION(println)(_UC_PRIMITIVE(string)("Hello world!"));
    }
  }

} // namespace _uc

int main(int argc, char **argv) {
  _uc::_UC_ARRAY(_uc::_UC_PRIMITIVE(string)) args =
    _uc::_uc_make_array<_uc::_UC_PRIMITIVE(string)>();
  for (int i = 1; i < argc; i++) {
    _uc::_uc_array_push(args, _uc::_UC_PRIMITIVE(string)(argv[i]));
  }
  _uc::_UC_FUNCTION(main)(args);
  return 0;
}
```

First, there are some includes of library files written in C++, which define built-in uC functions and types as well as macros and templates to be used by the generated code. The generated code itself is placed in the `_uc` namespace to avoid clashing with other code. Forward declarations of types and functions come first, allowing them to be used before their definition as required by uC. Then there are full type and function definitions. The generated code uses macros to *mangle* names into identifiers that will not clash with C++ names or with each other, in the case of names belonging to different categories. Finally, a C++ `main()` function converts command-line arguments into the format expected by uC and calls the uC `main()` function.

The compiler is run as follows to generate C++ code:

```
> python3 ucc.py -C <source file>
```

This parses the source file, runs the semantic analysis phases, and generates C++ code and saves it to a file. You can then compile the resulting C++ code as follows:

```
> g++ --std=c++14 -I. -o <executable> <generated file>
```

The `-I.` argument tells `g++` to search the current directory for header files and is necessary if the generated file is not in the current directory.

For the example above, we can run:

```
> python3 ucc.py -C tests/hello.uc
> g++ --std=c++14 -I. -o hello.exe tests/hello.cpp
> ./hello.exe
Hello world!
```

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

Distribution Code

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

File or directory	Purpose	What you need to do with it
`ucc.py`	Driver	Read it
`defs.h`	Library	Read and use it
`ref.h`	Library	Read and use it
`array.h`	Library	Read and use it
`library.h`	Library	Read it
`expr.h`	Library	Read, use, and modify it
`Makefile`	Testing	Run it with `make`
`tests/`	Testing	Read, use, and modify it
`life.uc`	Testing	Read, use, and modify it

You will also need to use and modify your code from Project 4. You will specifically need to modify `ucbase.py`, `ucstmt.py`, and `ucexpr.py`. You will also need to modify the `ucbackend.py` file included in the Project 4 distribution.

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 `-S` command-line argument is present, it only performs semantic analysis on the AST without any code generation. Otherwise, if the `-C` argument is present, it disables all but the first two semantic-analysis phases and invokes the code-generation phases the AST to generate C++ code to a file.

The interface for the backend compiler phases is defined in `ucbackend.py`. If all backend phases are enabled, the distribution code calls `gen_header()` to generate the header for a generated program, which includes library files and opens the `_uc` namespace. Then each backend phase is invoked in turn:

1. `gen_type_decls()`, which is responsible for generating forward declarations for each user-defined type
2. `gen_function_decls()`, which generates forward declarations for each user-defined function
3. `gen_type_defs()`, which produces definitions for each user-defined type
4. `gen_function_defs()`, which is responsible for producing definitions for each user-defined function

Then the distribution code calls `gen_footer()`, which closes the `_uc` namespace and generates a C++ `main()` function that calls the uC `main()`. You will need to place your code for invoking code generation on the AST within the top-level functions for each phase.

The `ucc.py` driver also provides a `--backend-phase` command-line argument to limit which backend phases run. For example, invoking `ucc.py` with `--backend-phase=3` runs only Phases 1 through 3 of the backend, without calling `gen_header()` or `gen_footer()`. Our testing framework makes use of this to test the early phases of code generation.

Unlike in Project 4, it is up to you how to structure your code for code generation. In the distribution, `ASTNode` defines an `emit()` method. However, it isn't called from by the distribution code. You may choose to call it directly, or you may decide to add other member functions in `ASTNode` that are then called from the top-level functions for each phase.

A `PhaseContext` object, defined in `uccontext.py`, provides several methods to print to an output file. The distribution sets the output file appropriately for code generation, and you should use the print functions when generating code. At the beginning of code generation, indentation is set to two spaces. You may wish to change the indentation level at various points to produce more readable output code.

The following print functions are implemented in `PhaseContext`:

- `print_noln()`: print arguments to the output file, but without a trailing newline
- `print()`: print arguments to the output file, with a trailing newline
- `print_in_noln()`: print the current indentation followed by the arguments to the output file, but without a trailing newline
- `print_in()`: print the current indentation followed by the arguments to the output file, with a trailing newline

We recommend using Python 3 string formatting or Python 3.6 f-strings to construct output text.

Library Files

The header files provided in the distribution are C++ library files that are included in a generated uC program. They implement useful macros and templates that can be used by the code generator, as well as definitions for built-in uC types and functions. The preamble code generated by `ucbackend.code_gen()` includes the library headers from the output C++ file.

The most basic macro definitions are in `defs.h`. It defines *name mangling* macros that convert uC names to distinct C++ names. These are:

- `_UC_PRIMITIVE`: mangles the name of a primitive type
- `_UC_TYPEDEF`: mangles the name of a user-defined type, as used when generating the struct definition of the type
- `_UC_TYPE`: mangles the name of a user-defined type and wraps it in a `_uc_reference`, providing the reference semantics required for user-defined types
- `_UC_ARRAY`: denotes an array of the given element type, which must itself already be mangled
- `_UC_FUNCTION`: mangles the name of a function
- `_UC_VAR`: mangles the name of a variable

Make sure to apply the appropriate macro when generating code for one of the entities above. Within the uC compiler, `Type` and `Function` objects define a `mangle()` method that you can use.

The library file `ref.h` defines a `_uc_reference` template that implements reference semantics, as well as memory management using reference counting. It uses `std::shared_ptr` in order to perform the latter. Except for constructing a null reference, you will not have to create a `_uc_reference` object directly in the compiler. Instead, use the `_uc_make_object()` function template to allocate an object and wrap it in a reference. You will need to explicitly instantiate the template, such as `_uc_make_object<_type(foo)>(...)`. To construct a null reference of a specific type, use the default constructor for a `_uc_reference` of the appropriate type.

The file `array.h` implements operations on arrays. The function template `_uc_make_array()` constructs an array of the given element type. As with `_uc_make_object()`, you will need to instantiate it explicitly. The templates `_uc_array_length()`, `_uc_array_push()`, `_uc_array_pop()`, and `_uc_array_index()` implement the corresponding array operations.

The file `library.h` defines the appropriate aliases for primitive types as well as the built-in library functions. You will not have to use any of these directly.

The file `expr.h` is intended to implement function overloads for the polymorphic operations in Phase 5. You will need to fill in the code for `expr.h` and use it when generating code for the polymorphic operations. The `_uc_length_field()` function template obtains the `length` field from a user-defined uC object. You will need to add one or more overloads for obtaining the `length` field from an array. The `_uc_add()` overloads should add two items together. You will need to provide overloads for the combinations of types that may be added in uC. Do not repeat code; use function templates where possible.

Testing Framework

A basic testing framework is implemented in the `Makefile`. There are separate targets for each of the phases:

1. `make phase1`: Run only Phase 1 of code generation. Compile the output with a provided `*_phase1.cpp` test file using `g++ -c`, so that `g++` does not attempt to link the resulting object file.
2. `make phase2`: Run Phases 1 and 2 of code generation. Compile the output with a provided `*_phase2.cpp` test file using `g++ -c`.

3. `make phase3`: Run up to Phase 3 of code generation. Compile the output with a provided `*_phase3.cpp` test file using `g++`, producing an executable. Run the resulting executable.

4. `make phase4`: Run all phases of code generation on tests that only require up to Phase 4. Compile the output using `g++`, producing an executable. Run the resulting executable with command-line arguments `20 10 5 2`, save the output to a file with extension `.run`, and compare against the expected output in the `.run.correct` file.

5. `make phase5`: Run all phases of code generation on tests that require up to Phase 5. Compile the output using `g++`, producing an executable. Run the resulting executable with command-line arguments `20 10 5 2` and compare the results to the expected output.

You may need to change the value of the `PYTHON` and `CXX` variables if the Python executable is not in the path or not called `python3`, or if you want to use a different C++ compiler. You can do so from the command line:

```
> make PYTHON=<your python here> CXX=<your C++ compiler here> ...
```

Test Cases

The following basic test cases are provided in the `tests` directory, with output from running the resulting executable.

Test	Description
`default.uc`	Test default constructors for user-defined types.
`equality.uc`	Test equality and inequality comparisons.
`hello.uc`	Simple "hello world" example.
`length_field.uc`	Test accessing the `length` field of an array or an object.
`literals.uc`	Test literals and addition between strings and other types.
`particle.uc`	Complex example of a uC program.
`use_before_decl.uc`	Test using types or functions before their declaration, which should be valid.

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.

You will not be able to directly compile the generated C++ code from a uC program until you complete Phase 4, since you will need to be able to generate the uC `main()` function. However, for each test case, we provide phase-specific test files in the form of `<test>_phase{1,2,3}.cpp`. These files directly test the functionality required up to the associated phase. For Phases 1 and 2, they ensure that the generated declarations are correct. For Phase 3, the test files determine whether or not type definitions are correct. Use the provided `Makefile` to run these tests, as described in Testing Framework.

Phase 1: Generating Type Declarations

The semantics of uC allow types and functions to be used before their definition. In order to support this in the generated C++ code, *forward declarations* must be made for each user-defined type and function. In addition, forward declarations of types should be made before those of functions, since a function may name a user-defined type in its return or parameter types.

Thus, the first step is to generate forward declarations for user-defined types. As an example, consider the following uC type definition:

```
struct foo(int x, float y);
```

This should result in a C++ forward declaration as follows:

```
struct _UC_TYPEDEF(foo);
```

Since this is a forward declaration, this is not a definition for the resulting struct.

Modify `gen_type_decls()` in `ucbackend.py` so that it runs your code for this phase.

Phase 2: Generating Function Declarations

The second step is to generate forward declarations for user-defined functions. As an example, consider the following uC function:

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

This should result in a C++ forward declaration as follows:

```
_UC_PRIMITIVE(void)
  _UC_FUNCTION(main)(_UC_ARRAY(_UC_PRIMITIVE(string)) _UC_VAR(args));
```

The return and parameter types should be mangled appropriately. You will find the `mangle()` method of `Type` objects useful for this purpose. The parameter names are optional in a forward declaration, but if you choose to generate them, make sure to mangle them with the `_UC_VAR` macro.

Modify `gen_function_decls()` in `ucbackend.py` to run your code for this phase.

Phase 3: Generating Type Definitions

Full type definitions must appear before function definitions, since in C++, the contents of a class or struct are not accessible until after the definition. Thus, the next step is to generate definitions for each user-defined type.

In creating an instance of a user-defined uC type, it is legal to provide a single argument for each field, or to provide no arguments at all. The latter results in default initialization, which initializes primitive types to 0, false, or an empty string, and references to user-defined types to null. Consider the following type:

```
struct bar(int x, foo f);
```

Both of the following initializations are valid:

```
b1 = new bar(3, new foo(4, 5));
b2 = new bar();
```

The former explicitly initializes each field to the corresponding argument, so that `b1.x` is 3 and `b1.f` is a reference to the newly created `foo` object. The latter performs default initialization, resulting in `b2.x` being 0 and `b2.f` null.

You will need to support both possibilities for initialization. Default initialization in uC is equivalent to value initialization in C++. You will need to generate code to value initialize fields in cases that require default initialization from uC.

User-defined types also must support equality comparisons. You will need to overload `operator==()` and `operator!=()` to do member-by-member comparisons. Use a signature like the following, so that comparisons can be done on r-values, which bind to `const` l-value references in C++:

```
struct _UC_TYPEDEF(bar) {
  _UC_PRIMITIVE(boolean) operator==(const _UC_TYPEDEF(bar) &rhs) const {
    ...
  }
  ...
};
```

Modify `gen_type_defs()` in `ucbackend.py` to invoke the code for this phase.

Phase 4: Generating Function Definitions

The next step is to generate full definitions for each user-defined function.

Make sure to place declarations of local uC variables at the top of the body of a generated function, taking care to mangle them with `_UC_VAR`. You do not need to initialize local variables, since uC specifies that their initial values are undefined.

Most uC statements and expressions have a one-to-one correspondence with C++ statements and expressions, and in those cases, you'll find that you'll be able to generate C++ code that is nearly identical to uC code. The semantics of most uC expressions, in terms of their order of operations and their types, are designed to be identical to their C++ counterparts.

For string literals, make sure to convert the literal text to a uC string, so that C++ does not consider it a `const char *`:

```
_UC_PRIMITIVE(string)("Hello world!")
```

For call nodes, the `func` attribute will **not** be available, since we cut off semantic analysis after the first two frontend phases. Instead, mangle the function name using the `_UC_FUNCTION` macro.

For allocation nodes, you should make use of the library templates `_uc_make_object<T>()` and, for array allocations, `_uc_make_array<T>()`. You will need to explicitly provide the first template argument, which should be the mangled name of the type or element type, when calling them.

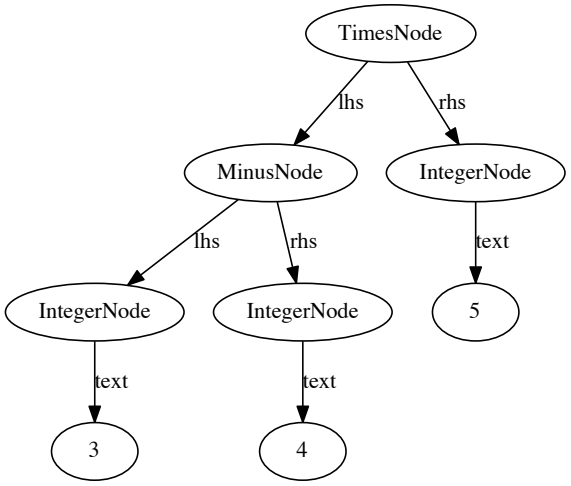
For field accesses, the receiver is always of `_uc_reference` type. Since `_uc_reference` derives from `shared_ptr`, it supports the indirect member-access operator `->`, which you can use to access a field in the general case. However, for the specific case where the field name is `length`, the receiver may be of object or array type. We will handle that case in Phase 5.

For array indexing, you will find the library template `_uc_array_index()` useful.

For unary and binary operations, you should place parentheses around the expression in order to preserve the precedence and associativity encoded in the structure of the AST. For example, consider the following uC code:

```
(3 - 4) * 5
```

This results in the following AST structure:



The parenthesization is encoded in the structure of the AST itself, so that 3 and 4 are grouped together under the `MinusNode`. In order to preserve this in the generated code, emit parentheses around every binary and unary operation (that has a C++ counterpart):

```
((3 - 4) * 5)
```

In uC, addition is a polymorphic operation, as it can be applied to numerical types as well as strings. Thus, its implementation is deferred to Phase 5.

For array push and pop operations, you will find the `_uc_array_push()` and `_uc_array_pop()` library templates useful.

Modify `gen_function_defs()` in `ucbackend.py` to execute the code for this phase.

Phase 5: Polymorphic Operations

The final compiler phase is to write implementations for two polymorphic operations, accessing the `length` field of a receiver and adding two items.

In a field access, the receiver may be of object or array type if the name of the field is `length`. In order to support this, we implement library overloads for `_uc_length_field()` and rely on the C++ compiler's overload resolution to select the correct one. We provide an overload for object types in the distribution file `expr.h`. Your task is to write the overload for array types. You should then generate code to call `_uc_length_field()` when the field name is `length`:

```
args.length
```

This results in:

```
_uc_length_field(_UC_VAR(args))
```

An addition is also polymorphic, as the operands may be both of numerical type, both of string type, or one of string type and the other of numerical or boolean type. To support this, define overloads for `_uc_add()` in `expr.h`. Then generate code to call `_uc_add()` for an addition. You may find `std::to_string()` useful for converting numerical types to strings. However, you cannot use it for booleans, which should be converted to the strings `true` or `false`, not `1` and `0`. Instead, you will need to define specific `_uc_add()` overloads for booleans that perform the correct string conversion.

Do not repeat code if not necessary for the overloads in this phase. Instead, use templates where possible. For reference, our solution has two overloads for `_uc_length_field()`, including the one provided in the distribution, and six for `_uc_add()`.

You may run into a case where the C++ compiler considers a call to be ambiguous because two or more overloads are equally applicable. In such a case, you can resolve the ambiguity by providing a more specialized overload that will be preferred over the ambiguous ones.

Phase 6 (Optional): Writing a uC Program

Implement a simulation of Conway's Game of Life in uC. As with HW1, your implementation should be on a finite grid. Edge cells have fewer neighbors but should otherwise follow the same rules as any other cell.

Complete the implementation of the `simulate()` function in `life.uc`. You may define any helper functions and structs you need.

The output format is the same as in HW1. When printing the grid, you should first print a line consisting of $cols+2$ dashes (`-`). Each row should then start and end with a pipe (`|`), each live cell should be printed as an asterisk (`*`), and each dead cell should be printed as a space. Finally, print another line consisting of $cols+2$ dashes followed by another empty line.

We have provided a test case in the `main()` function and the expected output in `life_test.out`. Compile and run your code with:

```
> python3 ucc.py -C life_test.uc
> g++ --std=c++14 -I. -o life.exe life.cpp
> ./life.exe
```

Alternatively, you can use the `Makefile` to compile and test `life.uc`:

```
> make life
```

This phase is optional and will not be graded.

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 uC constructs, so you should write extensive tests of your own.

We will evaluate your compiler based on whether the generated code is a valid C++ translation whose behavior matches that expected from the uC source. Thus, we will run a valid uC source file through your compiler to produce the output C++ code. For Phases 1, 2, and 3, we will combine your output with our own C++ test code that relies on the output being correct. For Phases 1 and 2, we will compile with `g++ -c`, without linking, to ensure that the declarations are correct. For Phase 3, our test code will use assert statements to test that your generated type definitions are correct. For Phases 4 and 5, we will compile your generated code with `g++`, and run it and check that the output matches what is expected.

Your generated C++ code itself does not have to exactly match ours. However, its behavior must match the behavior of our C++ code.

We will not test your compiler with erroneous uC code.

Grading

The autograded portion of this project will constitute the entirety of your project score. We will not hand grade this project. However, we are still requiring that you submit your test cases.

Submission

All code that you write for the interpreter must be placed in the files `ucbackend.py`, `ucbase.py`, `ucexpr.py`, `ucfunctions.py`, `ucstmt.py`, `uctypes.py`, or `expr.h`. We will test all seven 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 all of `ucbackend.py`, `ucbase.py`, `ucexpr.py`, `ucfunctions.py`, `ucstmt.py`, `uctypes.py`, `expr.h`, and any of your own test files to the autograder before the deadline. **You must submit to the autograder -- pushing code to GitHub does not submit it to the autograder.** We suggest including a `README.txt` describing how to run your test cases.