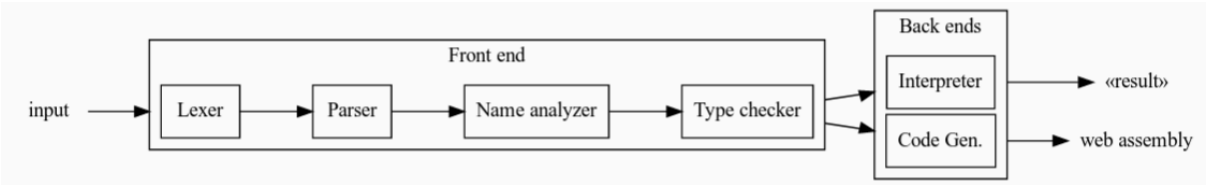
Extension 5.5: Compiling Amy to C

Compiler Construction ’13 Final Report

Violeta Vicente Cantero, Alexander Mueller, Jonas Bonnaudet
EPFL
violeta.vicentecantero@epfl.ch alexander.muller@epfl.ch
jonas.bonnaudet@epfl.ch

1. Introduction

We constructed a compiler for the Amy language that compiles to WebAssembly. To do this, we started by building an interpreter as it is easier.



Then we followed this pipeline and built a lexer that creates a series of tokens, a parser that makes LL1 ASTs, a type checker and a web assembly code generator.

Note that there is no name analyzer.

Our extension consists in creating an alternative C backend, which allows us to output C code.

2. Example

This extension is useful to be able to write Amy code for any platform that supports C. This greatly expands the number of devices that Amy can be written for, including any computer running Linux and microcontrollers such as Arduino.

Here is an example of converting a factorial function in amy to C:

```
object Factorial
    fun fact(i: Int(32)):
Int(32) = {
    if (i < 2) { 1 }
    else { i * fact(i-1) }
```

```
void Factorial_fact() {
    int locals[1] = {peek
(1)};drop;
  getLocal(0);
  cnst 2;
```

```
  }

   Std.printString("5! = "
++
Std.intToString(fact(5)));
  Std.printString("10! = "
++
Std.intToString(fact(10)))
end Factorial
```

Factorial.amy

```
  a = pop; push (pop < a);
  if (pop){
    cnst 1;
  } else {
    getLocal(0);
    getLocal(0);
    cnst 1;
    a = pop; push (pop - a);
    Factorial_fact();
    push (pop * pop);
  }
}
```

Factorial.c

Here peek(n), drop, getLocal(n), etc are C macros.

3. Implementation

3.1 Theoretical Background

Some basic WebAssembly concepts we had to look up and clarify were the WebAssembly modules that were being created in the ModulePrinter file. They are just objects that contain stateless WebAssembly code that has already been compiled by the browser. As we were just replacing WebAssembly code with C equivalents, we just had to understand what was being done and write it in C.

WebAssembly locals getting created at every function call was also something we had to take into account. We simulated that functionality in our implementation too and used them the same way.

We made heavy use of C macros in our implementation (*Replacing Text Macros*, 2022). They work very similarly to search and replace but they can also take function-like arguments.

3.2 Implementation Details

For our implementation, we replaced the Web Assembly code generation with C equivalents. In order to achieve this, we replaced the Web Assembly header code with C code that created a “virtual environment” that included an array of globals, a stack, and a stack pointer. We created a couple of macros here too, such as pop and push for the stack.

Then in the code generation section, we were simply able to replace all operations

with C operations, such as `addi32` by `push (pop + pop)`. There was a quirk; for non-commutative operators such as division, Web Assembly takes the top value from the stack as the second operand, so pop / pop would result in the inverse of the answer. So, we created a global variable in C called “a” to pop a value without using it. This division is represented as `a = pop; push (pop / a)`. It is not possible for a to cause a name collision, as the generated C code never creates any named variables from the Amy code, just as Wasm.
Function generation also has some quirks. All the generated C functions return void, since Web Assembly functions “return” values by pushing to the stack. They also accept no parameters, since these are also loaded from the stack. The first line of each function is the creation of a locals array. If statements are easy, since they are exactly the same between Web Assembly and C.
Lastly, we also redefined Amy's standard library in C. For instance, the standard print function was just a call to printf in C, since both languages represent strings as a location in memory with a null byte at the end.

4. Possible Extensions

4.1 Use the tests in lab5 to verify our implementation.

Lab 5 had a lot of input/output tests that could be reused with our C code generation because it only tests the output of the generated code and not the output of the code generator.

4.2 Dynamic reallocation of memory, globals, stack, as needed.

Our version of the code generator is a prototype and doesn’t implement any form of dynamic allocation of memory. Instead it allocates a fixed amount of memory on execution which is not optimal. The size of the stack can actually be computed at compile time as discussed during the presentation.

4.3 Finish implementing all standard functions from Std.amy.

Some functions in Std.amy need functions from the C/Wasm library and thus need special treatment at compile time. Some are not used in the tests so we didn’t implement them. To make the extension complete the functions need to be implemented and have tests.

4.4 Make it seamless to generate code for arduino.

An interesting extension to our code generator is arduino code generation. It is already possible to run the generated code on an arduino with some small performance tweaks such as removing unused functions, lowering the stack size and renaming the main function to setup.

References

Replacing text macros. (2022, December 30). cppreference.com. Retrieved January 8, 2023, from https://en.cppreference.com/w/cpp/preprocessor/replace