Wacc.wacc

A self hosting Wacc compiller

Nixon Enraght-Moony

Short Contents

Table of Contents

1 An overview of the Wacc

If your reading this, you probably already framiliar with the Wacc language. Hence this will not be a introduction to the language, nor a formal specification. Instead it is an overview of relavent pieces to building a self hosting wacc compiler. For a formal specification of the language, see the spec (`https://gitlab.doc.ic.ac.uk/lab2122_spring/wacc_examples/-/raw/master/WACCLangSpec.pdf`) and the examples repo (`https://gitlab.doc.ic.ac.uk/lab2122_spring/wacc_examples`).

- Incomplete Specification

  The Specification given for Wacc is incomplete. this version (`https://gitlab.doc.ic.ac.uk/rd3918/wacc_examples/-/raw/master/The%20WACC%20Language%20Specification/WACCLangSpec.pdf`) is what See Chapter 5 [wacc-to-c], page 6, links too, but in time my own verion will be made.

- Bad strings

  Wacc has very limited string support. They can do

  - `print` and `println` them
  - Be constructed by a litteral.

  Thats all! So the only use for strings is fixed strings for output. Most of the "string" handling code must use `char[]`

1.1 Extensions

A key part of the "Wacc Experience" ® is designing and implementing extensions to the language. Wacc.wacc attempts to not make use of these, mainly to prove a point, but also to make it easier to implement, at the cost of making it harder to implement.

One exception is the "extern" extension, which allows calling arbitrary C functions (provided their ABI can be expressed in the unextended wacc type system). This is nessisary for reading input (See Section 6.1 [input], page 7). However to keep this modular, to allow for alternate IO extensions to be used to bootstrap, only getchar (`https://www.gnu.org/software/libc/manual/html_node/Character-Input.html#index-getchar`) is used. It is declaired as `extern int getchar()`, and we assume `EOF` is -1 and C's `int` is the same as wacc's `int` (32 bits each), which is true on most modern systems.

Their are many extensions that I would like to use, but havn't, as they would increase the work to implement them, and would make this less of a "self hosting wacc compiller", and more of a "self hosting wacc++ compiler", even though it could be more accutatly described as a "self hosting wacc with extern compiller".

- Arbitrary Expressions: Don't have special items in <assign-rhs>.
- Switch statements
- Agragate types
- C style enums / C++ enum classes
- Rust style enums
- IO Library
- Module system

- More flexible if statement (eg if without else, else if without several fi's)
- Dynamic array allocation
- Garbage Collection
- String and array slices
- Built in generic higher level types (Go's slice and map)

2 Limitations

Due the the idiosyncracies of WACC, and the fact that writing something that works at all is hard enough, wacc.wacc has several limitations over a "fully complient" Wacc compiler.

2.1 Lack of checks.

Wacc.wacc will assume that the input is valid an not check it. This includes, but is not limited to

- Functions existing
- Function arity
- Function argument types
- Function return types.

2.2 Input file size

Wacc.wacc can handle at most L "logical lines" of input, where each logical line has has at most C characters. For more info See Section 6.1 [input], page 7.

These parametes are easily changed with the `NUM_LINES` and `NUM_CHARS` constants, in `py/gen_line_alloc.py`. Currenty they are `NUM_LINES = 3_000` and `NUM_CHARS = 400`. As the compiller is written, these will be increased to handle the larger input. I can't imagine anyone writing a program larger than the compiler, but if you do please let me know!

Longer input that this will do something, possibly undefined.

3 The build system

4 BS2: A Expiramental compiller

5 Wacc-To-C: A Bootstrap compiller

6 Lexing

Other than reading input, lexing is relativly trivial. The lexer is based on the clox scanner (`https://craftinginterpreters.com/scanning-on-demand.html`), by Bob Nystrom.

6.1 Reading input

Reading input in wacc is suprisingly hard. The only build in way is the `read`, and it only supports integer and character IO. Additionaly these only read after a newline is entered, and their is no way to detect an EOF, making it (AFAIKT) unsuitable for reading input. Therefor getchar(3) is used. See Section 1.1 [extensions], page 1, for how this is done.

The next chalenge is to read all of stdin. The problem with this is that to allocate a char buffer to read n charecters, you need to allocate $4n$ bytes (`' ',`). This is a problem for the compiler self hosting, as the buffer wouldnt be large enough to read the bytes that create the buffer.

Therefor, instead of reading stdin to `char[]`, it is read to `char[][]`, where each line has its own buffer allocated. Becauase array literals are secretly `mallocs`, we can have one literal for the list of lines, and one literal for a single line, which is shared amound lines.

However, we still want each token to only be on one line buffer. Most of the time using One line buffer per real line is fine, but not for multi-line string and char literal. Therefor ‘io::read_all‘ keeps track of if it is in a string (including escapes), and only switches to a new line buffer on a new line not in a string/char lit. This is the cause of “logical lines” in Section 2.2 [input-file-size], page 3.

Finaly memory management is a little tricky. ‘io::read_all‘ takes 2 sentinal charecter arrays, both of which should be empty. 1 is the default empty line, and the other is just used to indicate the end of input. This allows freeing all non empty line buffers, as they are unique, and then freeing the 2 sentinals.