



wacket




a lisp inspired functional language compiled to WebAssembly

Stefan “fun-call” Heller



Christopher “pretty-printer” Olin

Peter “hot-tub” Geertsema

Introduction

Partway through this semester, the wacketeers set out on a mission. We wanted to see how far we could push the boundaries of compilers. The end result was  wacket, which compiles to WebAssembly (WASM) (with an intermediate representation in WebAssembly text format (WAT)). Web Assembly is a new (released March 2017) programming language for web development, designed for performance-sensitive tasks on webpages. It is intended to be used in-tandem with JavaScript, and runs in the browser — eliminating any concerns about compatibility with various operating systems. As a result of its nascency and its security concerns, WASM is lacking many critical or helpful features, such as direct access to the stack and tail calls which  wacket is (mostly) able to work around. For more information about this project or how to get started writing and compiling  wacket, see the README within the project submission.

Features

 wacket was built to emulate the functionality of some of the most advanced programming languages we implemented in class. Every feature through Loot has been implemented, with the exception of Jig — tail calls (since WASM does not yet support tail calls), and Knock — pattern matching. Since  wacket supports lambdas, it is able to emulate many of the more advanced features of modern languages.

Who did what

“fun-call”	“pretty-printer”	“hot-tub”
Frontend	wat pretty printer	Fraud
Runtime System	Abscond	Hustle
Abscond	Blackmail	Hoax
Blackmail	Con	Provided a hot tub
Con	Dodger	
Dupe	Evildoer	
Extort	Extort	
Iniquity	Loot	





It should be noted that just because “hot-tub” is only credited with three items, by no means does that mean he did less work than the others. The features he implemented were some of the trickiest, especially considering the idiosyncrasies of WebAssembly, which came with an enormous learning curve. Many of the features implemented by “fun-call” and “pretty-printer” (while they were not easy) were more straightforward to implement with more of a base level understanding of WebAssembly.

Additionally, individuals were credited for features they took the lead on, but oftentimes particular features were collaborative efforts between some or all members of the group.


Implementation Specifics


It would not make sense to talk about what we changed for the project, because we changed virtually everything. The following are some of what we consider to be the bigger feats of the project.


Runtime System

While in the future WASM will ideally be platform agnostic and able to interface well with any runtime system, at the moment it is best to do everything in JavaScript. The way this works is that JavaScript has access to a WASM API which allows the runtime system to interact with the generated WASM module. This relationship goes both ways. WASM can import functions defined in the runtime system, and the runtime system can import functions defined within the WASM code.  wacket relies solely on importing a few functions from the runtime system, and not the other way around. While the JavaScript runtime system is mostly a 1:1 port of the C runtime system from Loot, it does have some additional functionality and requirements. In order to create an actually usable and understandable output from our  wacket code, we actually needed to have a front end for our language which allows for input and output displayed in HTML as opposed to printing to standard output as done in class. Our runtime system implements all the same type checking and conversion done in loot, but also includes an `error` function which throws a JavaScript error if it encounters an error. It also implements `readbyte`, `writebyte`, and `peekbyte`, which allow  wacket to have IO, as WASM on its own is incapable of that. Our runtime system also provides our  wacket program with its pseudo stack and heap in the form of a JavaScript array. WASM has no access to system memory, so it relies on JavaScript passing it an array to store heap objects in, and our own stack which is described below. The final component of our runtime system is the `run` function, which handles reading the input in our input box, outputting to the output box, and running our program on the click of the `run` button. In the future we hope to implement ways of visualizing programs and types that we don't yet have functionality for, such as allowing for the visualization of returned lambdas and other structs stored on the heap.

Heap/Stack


WASM is able to access memory which is allocated through the JS runtime system. JS is able to allocate memory in pages of 64KiB (addressable in JS as an array of 32 bit integers).  wacket uses two such pages, one for heap objects, and another for our implementation of the stack.

Why create our own stack when WASM has its own? Well I'm glad you asked, intrepid reader. WASM currently does not support local variable scoping, so  wacket needs to implement its own solution. WASM also only lets you access the value(s) at the top of its stack, so we needed to engineer a way to load a value arbitrarily down the stack, just like in the a86 based languages we designed in class. The resulting system has a page of memory allocated to the stack, and the other page for the heap.

One insidious bug I ran into was related to nested cons, where the `car` and `cdr` cells would not be placed together. This was due to nested cons cells being compiled before both cells had their space allocated. To solve this, I modified the  wacket AST struct for storing in the heap to support values already stored on WASM's stack. This meant I could safely grant memory for both the `car` and `cdr` without the nested structure of either cell interfering with the consecutive allocation of memory blocks.

Lambdas

Lambdas presented a unique challenge to the wacketeers. In Loot, proc structs on the stack reference the address of the start of the lambda's body's code. In WASM, however, the functionality of jumping to arbitrary sections of code referenced by address is not supported (for security reasons). Instead, calls to functions known only at runtime use a different instruction, `call_indirect`, which looks up a function's name by index in a special table, specified at the beginning of the program.

The challenge here is not that putting function names in the table is hard, but rather that each `call_indirect` must be accompanied by a function signature defined before the body of the program. This means that any possible combination of parameters and return values must be accounted for at the top level of the program (for example, two i64s to one i32, three i64s and an i32 to two i64's, etc.). For this reason, until further notice, **we have restricted  wacket's functionality regarding lambdas to only accept a single argument, and return a single argument.** The same does not apply to regular functions, as those are referenced directly by name, and are accessed by `call` instead of `call_indirect`. In the future, we intend to parse the program and determine the signatures of all lambdas used in the program, so that all signatures that exist may be accounted for before the body of the program.

Future Work

Some items we intend to tackle in the future have already been pointed out: RTS support for displaying heap structures and allowing lambdas to have arbitrary signatures. In addition to these items, we intend to implement features included in the class assignments, like case, cond, arity checking, and functions that accept arbitrary numbers of arguments. We also intend to implement Knock (pattern matching), which we did not prioritize work on because it came after Loot in the class schedule. Tail recursion is a feature we are interested in, but with the limitations of WebAssembly in mind, this feature will be put on the backburner for some time. Finally, due to the time constraint of this project, we feel that our code base is not particularly clean or well-documented. In the future, we plan on fixing this by adding or updating documentation, and rewriting or moving sections of code to be clearer.