Introduction

JavaScript is a client-side web scripting language. Usually, web browsers interpret JavaScript, or compile it into bytecode and then run the bytecode. I set out to write an extension to the Mozilla Firefox web browser that would compile JavaScript into native machine code and then run this code. It turned out to be a more ambitious project than I had anticipated and probably should have taken two semesters, but I was able to finish most of the programming and design how the system would work.

The extension has many parts. To compile JavaScript code, Firefox calls a function that causes a program written in ML to compile this code (this compiler is where most of the work for this project is) and produce an assembler file. This assembler file is then assembled and turned into machine code. Then, Firefox dynamically links the machine code, by basically copying it into memory. It then runs the code from memory.

Background on JavaScript and web browsers

For background about ECMAScript, how web browsers run JavaScript, Mozilla Firefox 3, TraceMonkey, and how this project fits in, see the original 490 overview.

Overview of the project

A webpage contains JavaScript code. When Firefox sees this, it causes Firefox to run its JavaScript interpreter. We have added a function called "compileRun" to the JavaScript interpreter.
When a user calls this function with a string, the browser calls the ML interpreter. ML runs a JavaScript compiler on this string of code that it has been passed. Then, the compiled code is assembled into machine code. This machine code is dynamically linked into Firefox, and Firefox runs this compiled Javascript code, which the user originally passed to the "compileRun" function.

What follows is a lower-level description of how the compiler works.

Lexer

In this stage, the JavaScript source string is converted into a sequence of lexical tokens. We mostly used the tokens described in the ECMAScript specification. We use the lexer generator for SML, ML-Lex.

Parser

Here, a sequence of tokens is converted into abstract syntax. Again, we mostly followed the ECMAScript specification, though with some workarounds. For example, supporting the "for...in" expression means making two entirely separate parse trees. We did not plan on implementing them later, so we skipped implementing them in the parser. Also, semicolons are dealt with in a funny way – ECMAScript treats newlines without semicolons as errors that are easily recovered from by inserting semicolons. In our version, we just require semicolons at the end of every line.

We use the parser generator for SML, ML-Yacc.

Type checker

In a statically typed language, this is where the compiler's type checker would be. The type checker assigns a type to each variable and statement. It also keeps track of the type and location (which frame) of variables, and allocates spaces for them on stack frames. But ECMAScript is dynamically typed, so that isn't here. See "Dynamic Typing" for more explanation and what we do
Intermediate tree generation

This is the heart of the compiler. This is where we convert abstract syntax - the programmer's idea of the programming language - into an intermediate tree generation - closer to the machine's idea of the instructions it needs to execute. Some code, like string literals, gets translated into raw data chunks, and the references to the string literals get translated to the assembly language label for where the raw data lives. We also did not implement everything completely here. We were able to use the tree code from Tiger, since we use the Tiger backend.

Backend

Here, the intermediate tree code from the phase before gets converted into assembly language. I was fortunate enough to have Lin He's and Glenn Thrope's backend from CS421.

Linking with Firefox

Once the assembly language is generated, we let the target machine assemble it into object code. The next step is to let Firefox run this code in its existing address space. To do this, we basically copy the code from its assembled form on disk into memory; make a function pointer that points to the code; and run this function. We do this with the dynamic linking loader, through the Linux functions dlopen, dlsym, and dlclose. Then, when this function runs, it runs the code that we compiled. The runtime functions that our code calls can now be resolved with functions that were we compiled into Firefox. (So for our project, we needed to modify the Firefox source to add new runtime functions for our code.)
Dynamic Typing

ECMAScript is dynamically-typed. That means that there is no type checker. Instead, all of this happens at runtime. So instead of the compiler keeping track of where variables are, it just translates a variable call with an assembly language "call" to a runtime function that resolves the variable (and maybe even allocates it). Also, the compiler doesn't know the types of expressions; at runtime, everything can be converted and coerced into different types.

Luckily, Firefox already has code to handle variables and coercion, so we were able to simply call that. We wrote a run-time wrapper for this code, called cGetPropertyLabel, in jsCompileAndRunTime.c. This function also handles scope resolution (see "Variables and Scope").

Firefox does have types, though. Every variable has a type: undefined; null; boolean; string; number; or object. This last type, object, is the type of almost every variable.

Variables and scope

In Javascript, every variable is resolved dynamically at runtime. As described above, "object" types are used very often. Their attributes, or children, are simply properties.

We handle scope the way that ECMAScript recommends: each level of scope has a different variable object, which itself is an object whose attributes are the current scope's local variables. Then, when a variable name is used in code, our "cGetPropertyLabel" function queries different variable objects (using Firefox's functions) and goes up the scope chain of different variable objects. If it can't find this variable name in any part of the scope chain, it makes a new variable in the current (lowest) scope.

Functions
Functions are first-class objects themselves. But they have a special (hidden, read-only) attribute that means that they can be called. And when the compiler sees that something is calling an object that has this attribute, it calls a new function in the same way other programming languages do – setting up a new stack frame, passing arguments, etc.

**Personal**

In retrospect, this project was too ambitious. I was not able to complete the entire JavaScript compiler in time. I probably should have made this a two-semester project. But despite that, it was possibly the most interesting project I have worked on at Yale. And I learned in a new and deep way about compilers and about how computers run code.

Some problems were logistical. For example, the Tiger compiler backend was for 32-bit machines, and Firefox was compiled for the 64-bit Zoo machines. I read online how to compile Firefox for a 32-bit system, but this required administrative access to install new programs. I logged on to a friend's computer and tried to do it there, but I couldn't get it to work. After trying many different things and spending about 7 hours, I gave up on this part.

But most of the hurdles were interesting and taught me about compilers. It took several days, and chats with Alex Vaynberg, to understand the idea of dynamic typing and how I would not need a symbol table or a type checker. Also, I struggled with the idea of local "variable objects" and what that meant for stack frames - but I realized eventually that I would still need a stack.

Now, each stack frame contains two things: the variable object on the current level; and a static link pointer (a pointer to the stack frame statically enclosing this one, used in resolving variables).

**Acknowledgments**

Thanks to Zhong Shao for his support. I chose this project because of how intense and
interesting I found his compilers class. Thanks also to Alex Vaynberg for his help. The backend of the JavaScript compiler is from Lin He's and Glenn Thrope's work in CPSC 421, fall 2007.

References


