Metaprogramming & Metacompilers

"Make Programming Great Again"

joe barnett - px8 solutions, inc.

joe's github

Overview

Metaprogramming Concepts

A ***Fantastic*** Langauge

Creating a Formal Grammar

Fundamentals of ***wasm***

Building a Compiler with Irony

Metaprogramming Concepts

What is "metaprogramming"?

I don't know.

```
Programs that abstract the concept of programming?
```

```
Programming = writing machine code by hand
Metaprogramming = everything else
```

```
Programs that are aware of themselves?
Programs that can modify themselves?
```

```
Programs that create other programs?
```

```
Metalanguage vs. Object Language?
Using English to discuss Mandarin?
A BNF for SQL?
```

All that.

- Assemblers/JITers (mnemonics/bytecode-to-machine code)
- Compilers (high-to-low) & Transpilers (high-to-high)
  - General Purpose Languages (GPL)
  - Domain Specific Languages (DSL)
- Macros
- Templates (simple)
- Polymorphic Programming
- Reflection (introspection)
  - .NET Attributes
  - Java Annotations

What is a "metacompiler"?

I don't know.

```
A compiler-compiler?
```

```
A tool that accepts a metalanguage and generates another tool
(a compiler) from that metalanguage that accepts another
language and generates yet another language?
```

Okay.

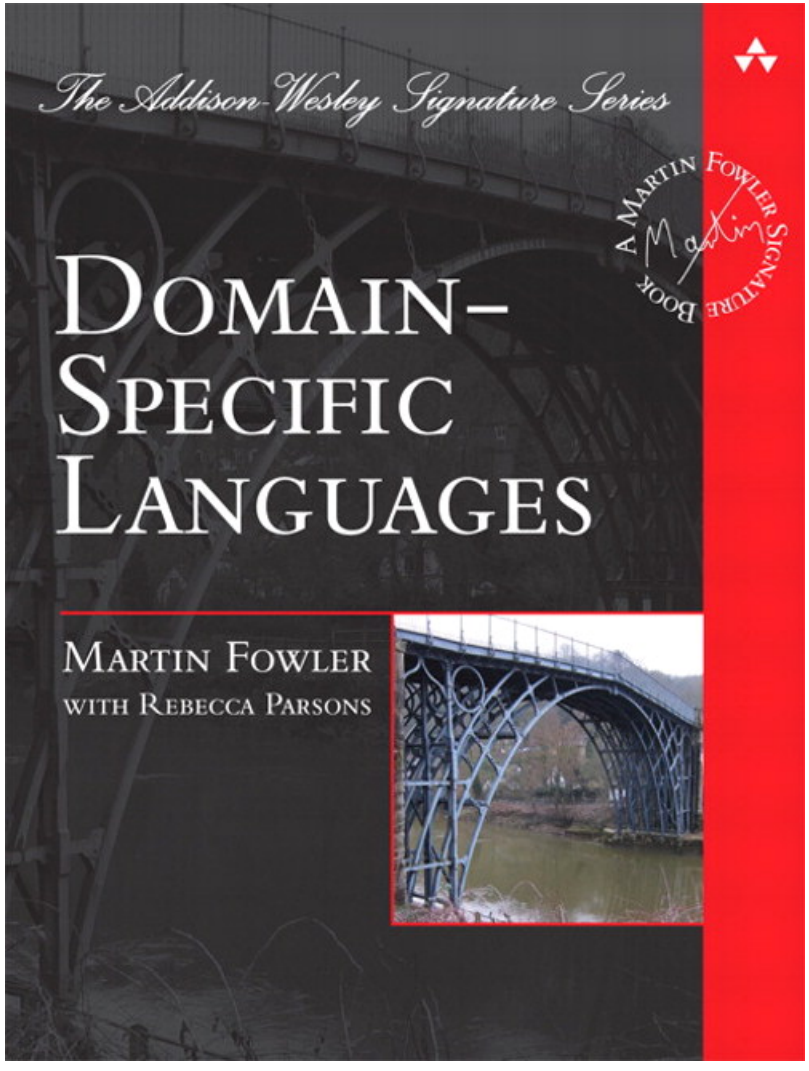
DSLs

- DSL vs GPL
  - DSL may be Turing Complete, but useful as a GPL?
    - "I know it when I see it." - Justice Potter Stewart
  - Captures domain knowledge that provides useful abstractions?
- Textual vs Graphical
- Internal vs External

DSL Examples

- External
  - SQL
  - HTML/CSS
  - BNFs for *compiler-compilers*
  - RegEx
  - LINQ Query Comprehension
- Internal (AKA embedded or minilanguage)
  - jQuery (all the "fluents")
  - LINQ Method/Lambda Syntax (a "fluent")
  - Active Record Migrations
  - EDMX (graphical)
  - Irony

The DSL Book


The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Martin
Domain-Specific Languages
Martin Fowler
with Rebecca Parsons

Macros and Templates

- Macros
  - text substitution/expansion
  - "macro" assemblers
  - C/C++ preprocessor
    - cpp in.txt -o out.txt
- Templates
  - T4
  - XSLT
  - Razor
  - ASP
  - JSP
  - PHP

Polymorphic Programming

- Ad-hoc (compile-time, not part of the type system)
  - Template Metaprogramming
    - C++
    - D
  - Java Generics (type erasure)
  - Function/Operator Overloading
- Parametric (run-time, reified in type system)
  - .NET Generics
  - ML
  - Haskell
  - Julia
  - Others

A ***Fantastic*** Language

Let's build a *totally awesome* general purpose language (sort of).

It will be *fantastic*.

(Anyone tired of Trump humor yet? No? *Awesome*!)

A *Fantastic* Language : CovfefeScript

Keywords:

```
trigger     - declare a trigger (function)
cuck        - declare a cuck (variable)
=           - dominate the cuck (assignment)
+++         - extreme addition
---         - very stable subtractions
bing        - increment
bong        - decrement
caterpillar - win forever (loop)
yerfired    - stop winning (break)
if          - low-energy conditional
==          - sameyness
shitpost    - print value
BTFO        - return from trigger
grab        - pop value from stack
//          - nasty comments
```

CovfefeScript Limitations:

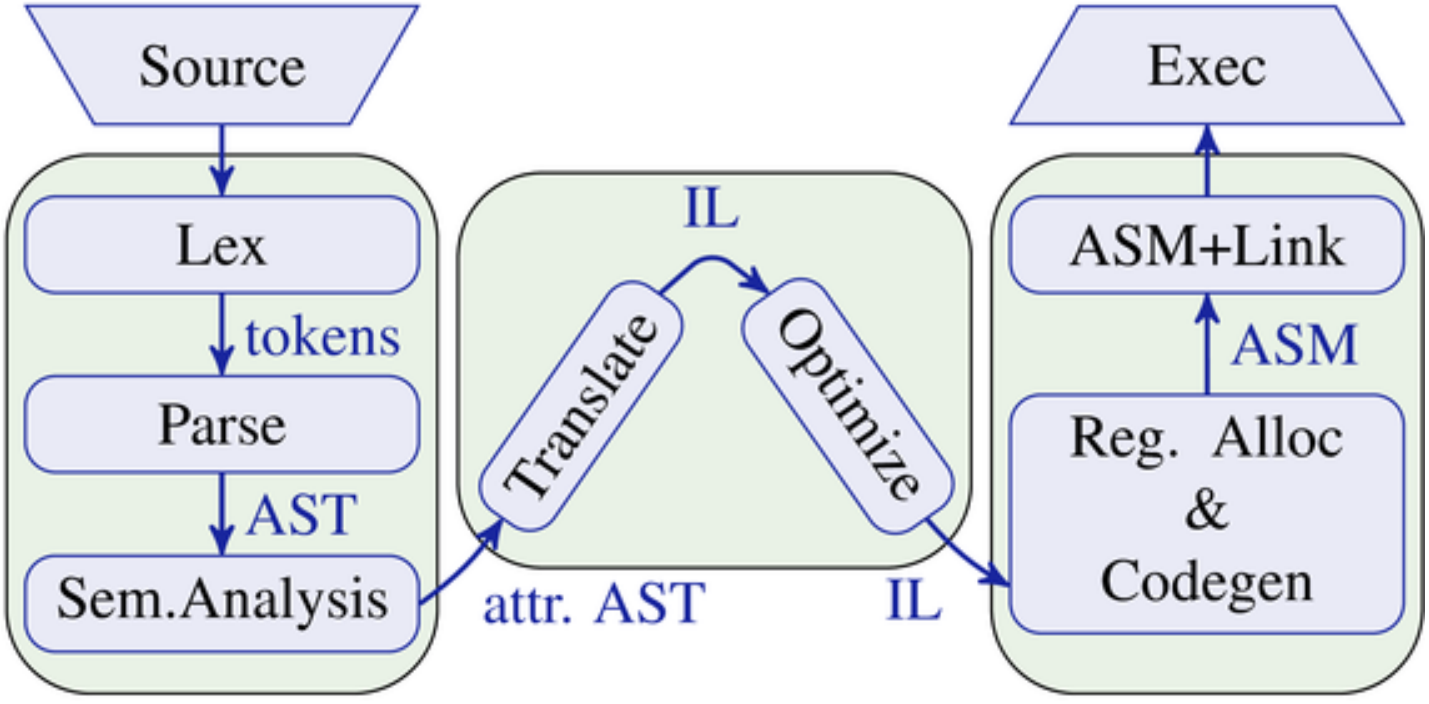
- One data type: *i32*
- add/subtract only
- equality test only
- No heap
- No exceptions!

But this all we need to win.

CovfefeScript Example

```
trigger main()
{
    cuck n
    cuck result

    n = 22 //253
    result = 0

    caterpillar
    {
        result = result +++ n
        bong n

        if (n==0)
        {
            yerfired
        }
    }

    shitpost(result)
```

How do we get started?



We'll need a grammar for our lexer/parser, and we'll use *wat* for our IL, and we'll need a translator, and we can use wat2wasm as the assembler, and we can... skip everything else.

Creating a Formal Grammar

"High level programming language"?

```
Alphabet: a set of symbols

        A = {a, c, o, t}
```

```
String: sequence of symbols

        cat
```

```
Language: a set of strings over an alphabet

        L = {cat, cot, sat} over alphabet {a, c, o, t}
```

We build *machines* or *automatons* to recognize or generate a specific language, following *rule*s.

Production Rules

A *grammar* is a set of rules transforming *non-terminals* (placeholders) into strings, given a set of *terminals* (an alphabet). A grammars defines a language (morphology/syntax). Grammars *generate* languages. Languages are recognized by *automata*.

```
1. S → aAb
2. A → ba
3. A → ε
```

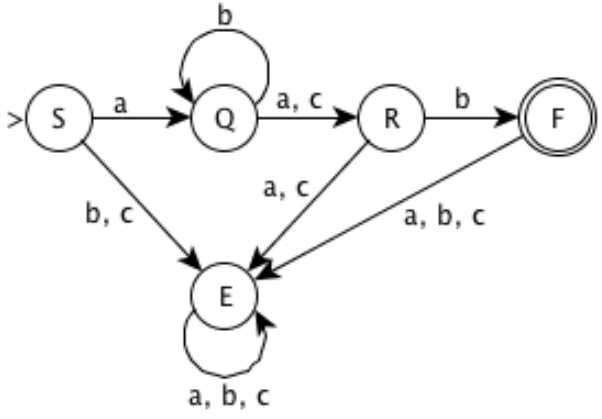
```
S → aAb → aaAbb → aababb
```

```
{ab, abab, aababb, aaababbb}
```

These ideas were developed by Alex Thue (1914), Emil Post (1930), Alan Turing (1936), and most importantly, Noam Chomsky (1956).

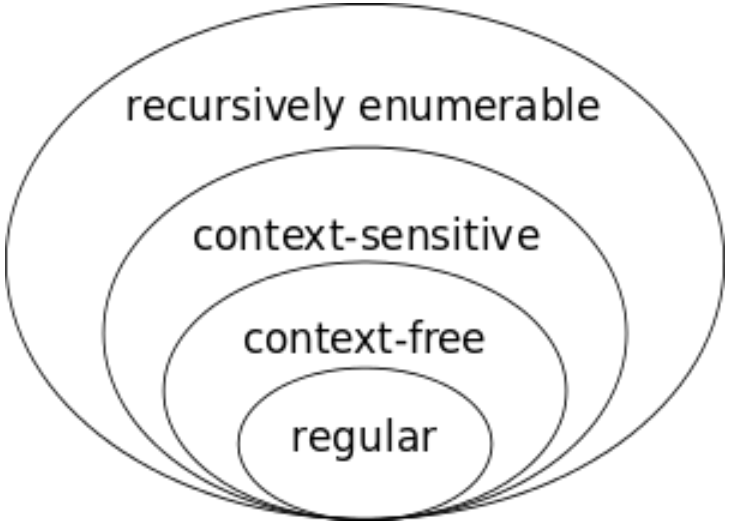
Languages have complexity

Simple languages can be rec/gen by simple machines. More complicated langauges require more complicated machines. We can use language to define ideas about computation and complexity.



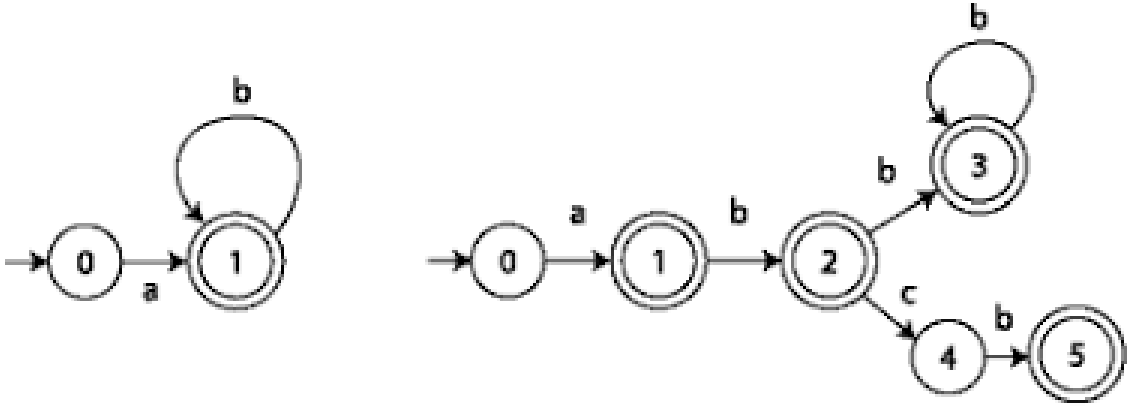
Different parts of the compiler (lexers and parsers) deal with different kinds of complexity.

The Chomsky Hierarchy




```
0 - recursively enumerable/unrestricted (turing machine)
1 - context-sensitive (LBA)
2 - context-free (PDA)
3 - regular (FSA)
```

Deterministic Finite Automaton

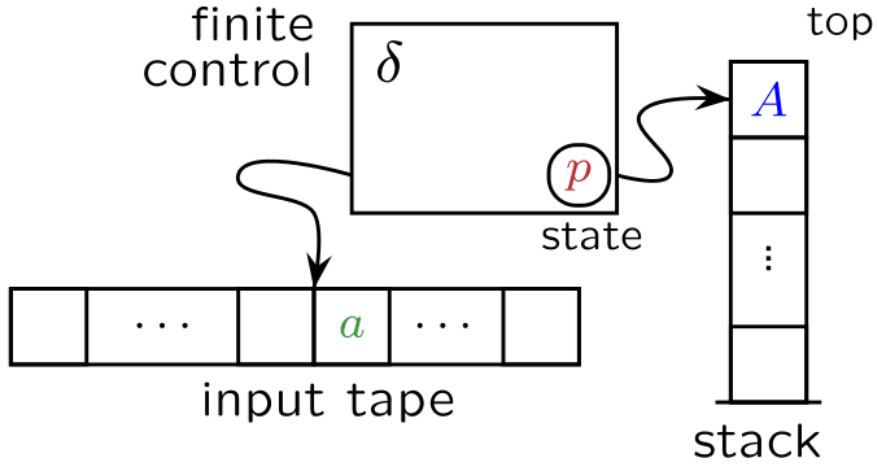


Regular languages (type 3). NDFA=DFA. Regular!=RegEx.

```
A → a
A → aB (right regular)
A → Ba (left regular)
A → Aa (this is okay)
```

Left-hand side has exactly one non-terminal, and the right-hand side has either a terminal or a single non-terminal with a terminal on the left or right but not both.

Pushdown automaton

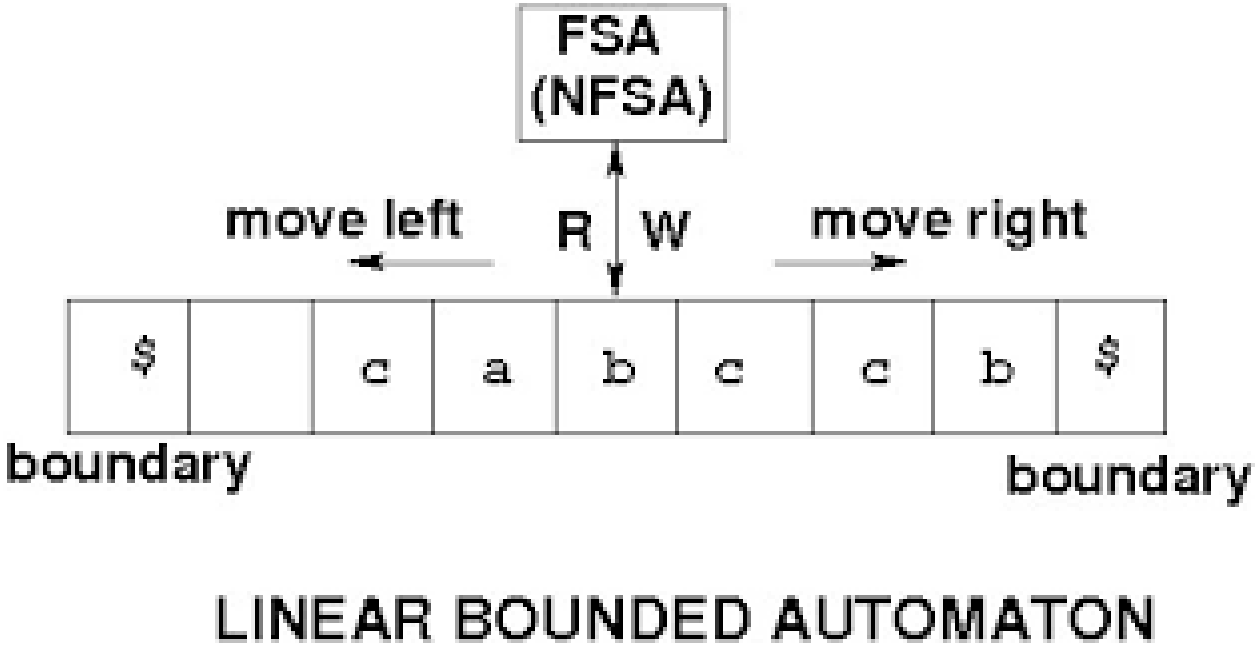


Context-free languages (type 2 - nesting). PDA (stack). DPDA!=PDA.

```
A → a
A → aBb
A → aAb
```

Same as above but now we have terminals on both sides of the non-terminal on the left-hand side. These grammars have nested structures (parentheses matching).

Linear bounded automaton



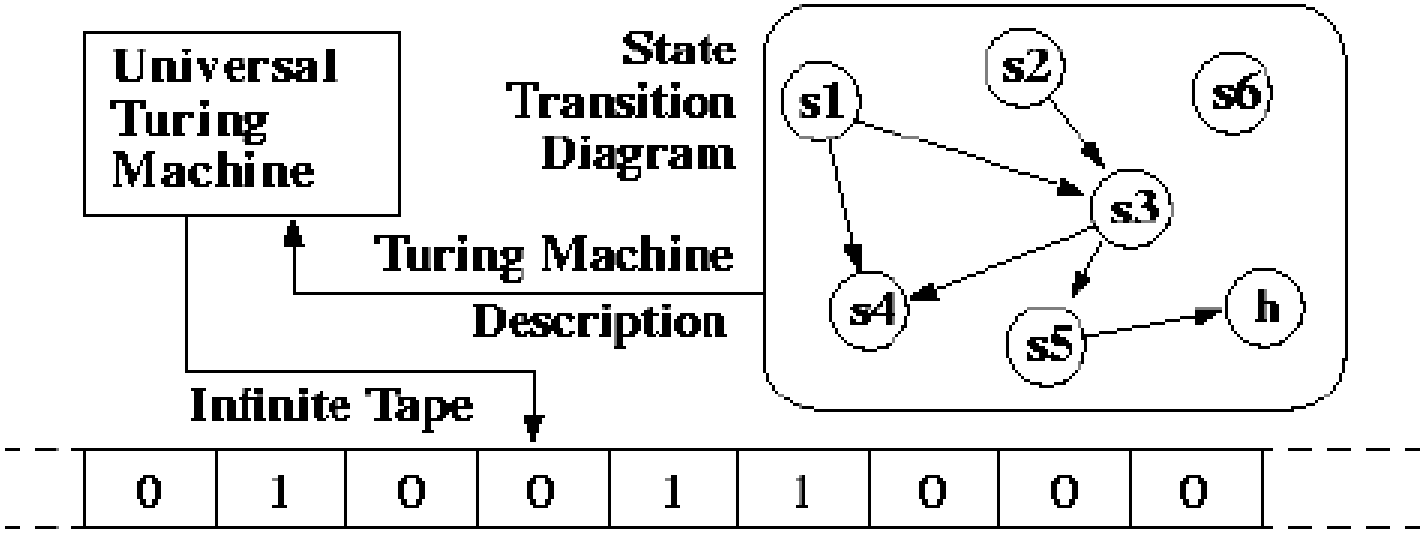
LINEAR BOUNDED AUTOMATON

Context-sensitive (type 1). LBA.

```
aAb → anything
CAB → anything
```

Not exactly but good enough for politics (leaving out epsilon).

Turing Machine



Recursively enumerable (type 0 - infinite but countable). Turing machine that halts when the language is recognized, may not otherwise. Recursive languages always halt.

```
anything → anything
```

Backus Normal Form (BNF)

A more convenient method of defining the syntax of programming languages was developed by John Backus (IBM) in 1959, BNF. This type of *metasyntax* captures the same information as the mathematical description used in formal grammars.

```
letter    = "A" | "B" | "C" | ...
digit     = "0" | "1" | "2" | ...
character = letter | digit | symbol | "_" ;
identifier = letter , { letter | digit | "_" } ;
```

```
statement = lhs , "=" , rhs , ";" ;
```

We usually use a fancier version, called *extended* BNF (EBNF).

CovfefeScript BNF - Terminals

Keywords:

```
trigger
{
}
cuck
grab
=
+++
---
bing
bong
shitpost
btfo
yerfired
if
...
```

CovfefeScript BNF - Terminals

Literals:

```
bin_digit = 0|1
dec_digit = 0|1|2|3|4|5|6|7|8|9
hex_digit = 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
alpha = A|B|C ... x|y|z
```

```
int = dec_digit* | "0x", hex_digit* | "0b", bin_digit
```

Note: this is a modified BNF for brevity on these slides.

CovfefeScript BNF - Terminals

Identifiers

```
alpha = A|B|C ... x|y|z
```

```
id = ["_" | alpha], ["_", alpha, dec_digit]*
```

Note: all terminals can be recognized by the *lexer* (DFA).

CovfefeScript BNF - Non-Terminals

Expressions (limited but *very* good)

```
OP = +++ | ---
VAL = [ id | int ]
EXP = VAL, OP, VAL
LVALUE = VAL | EXP
ARG = LVALUE
PARAM = id
```

Structure and statements:

```
CUCK = id | [ id, =, LVALUE ]
CALL = id, (, ARG*, )
BING = bing, PARAM
SHITPOST = shitpost, PARAM
...
STATEMENT = CUCK | CALL | BING | SHITPOST | IF | ...
TRIG = trigger, id, (, PARAM*, ), {, STATEMENT*, }
FILE = TRIG*
```

CovfefeScript BNF - Non-Terminals

These productions involve recursion:

```
TEST = [ ==, <, >, != ]
IF = if, (, LVALUE, TEST, LVALUE, ), {, STATEMENT*, }
CATERPILLAR = caterpillar, {, STATEMENT*, }
```

Note: it is the nesting of elements that promote this grammar to a *context-free* grammar, and now we'll need a proper parser to recognize our langauge (PDA).

CovfefeScript BNF

Okay, now that we have a BNF, we can build a parser, but what's our target?

wasm

Fundamentals of *wasm*

```
"WebAssembly or wasm is a new portable, size- and
load-time-efficient format suitable for compilation
to the web." - mozilla
```

wasm **is a specification:**

- stack-based virtual machine
- virtual instruction set architecture
- type system
- loadable image format (*wasm* modules)
- binary format (bytecode/bitcode)
- text format (instruction mnemonics in Lisp-like s-expressions)

Note: we'll be using the WebAssembly Text Format (*.wat) as the target for our CovfefeScript compiler, which we will then assemble into WebAssembly Binary Format (*.wasm) with the WABT toolchain.

wasm big ideas

these features:

- static typing
- low-level constructs
- compact binary format

allow these benefits:

- static program analysis (validate & optimize)
- improved download speeds
- near-native execution speeds
- universal compile target (more languages)
- improved reliability & security

Nope!

But we're not doing any of that today.

Instead, we're going to replace JavaScript once and for all with CovfefeScript. We can make it happen!

(See the emscripten project for more on that kind of stuff.)

wasm toolchain : components

- Git
- CMake
- C/C++ compiler
- Python 2.7.x
- WABT
- Compatible browser (most of them now)

Developer's Guide

wasm-by-hand

Understanding wast

Semantics

Reference

WABT : getting the source

Clone WABT (WebAssembly Binary Toolkit) source

```
git clone --recursive https://github.com/WebAssembly/wabt
```

Or

```
git clone https://github.com/WebAssembly/wabt.git
git submodule update --init
```

WABT : building

```
mkdir build

cd build

cmake .. -DCMAKE_BUILD_TYPE=DEBUG
         -DCMAKE_INSTALL_PREFIX=bin
         -G "Visual Studio 14 2015 Win64"

cmake --build . --config DEBUG --target install
```

Set your PATH

```
$ PATH=$PATH:"/c/wabt/bin"
```

WABT commands

- wat2wasm
- wasm2wat
- wasm-objdump
- wasm-interp
- wat-desugar
- wasm-link

Creating a module

Simplest *wasm* module (one line of *wat* assembly language):

```
(module)
```

The binary *wasm* output (8 bytes of *wasm* bytecode):

```
$ wat2wasm module.wat -o module.wasm
$ od -t x1 module.wasm
0000000 00 61 73 6d 01 00 00 00
0000010
```

```
0061736d - WASM_BINARY_MAGIC
01000000 - WASM_BINARY_VERSION
```

Binary specification

Creating a WebAssembly Binary

Creating a function

A complete module with a function in *wat* assembly language:

```
(module

  (func $add (param $p1 i32) (param $p2 i32) (result i32)
    get_local $p1
    get_local $p2
    i32.add
  )

  (export "add" (func $add))

)
```

Assembling & Examining

```
wat2wasm functions.wat -o functions.wasm
```

```
$ wasm-objdump functions.wasm -x

functions.wasm: file format wasm 0x1

Section Details:

Type:
 - type[0] (i32, i32) -> i32
Function:
 - func[0] sig=0
Export:
 - func[0] -> "add"
```

Disassembly

Reversing the assembly process to convert the binary *wasm* back to text-based *wat* assembl

```
$ wasm-objdump functions.wasm -d

functions.wasm: file format wasm 0x1

Code Disassembly:

000021 func[0]:
 000023: 20 00                          | get_local 0
 000025: 20 01                          | get_local 1
 000027: 6a                             | i32.add
 000028: 0b                             | end
```

wasm : Glue Code (printing)

```
var importObject = {
     imports: {
        print: function(arg) {
          var div = document.createElement("div");
          div.style.fontFamily = "courier";
          div.style.fontSize = "18pt";
          div.style.fontWeight= "bold";
          div.innerHTML = arg;
          document.getElementById("output").appendChild(div);
        }
     }
};
```

wasm : Glue Code (loading from server)

Load the *wasm* bytes and start a *main()* function:

```
var request = new XMLHttpRequest();
request.open('GET', 'hello.wasm');
request.responseType = 'arraybuffer';
request.send();

request.onload = function() {
  var bytes = request.response;
  WebAssembly.instantiate(bytes, importObject).then(
      function(result) {
        //result.instance & result.module
        result.instance.exports.main();
      });
  }
```

instantiate returns an ES6 promise. We can use Fetch API as well.

The File API should be possible too.

wasm : Glue Code (loading bytes)

Load the *wasm* bytes and start a *main()* function:

```
var bytes = new Uint8Array([
    0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, ...

 window.onload = function() {
    WebAssembly.instantiate(bytes, importObject).then(
      function(result) {
         result.instance.exports.main();
      });
 }
```

wasm : Glue Code (wiring it all together)

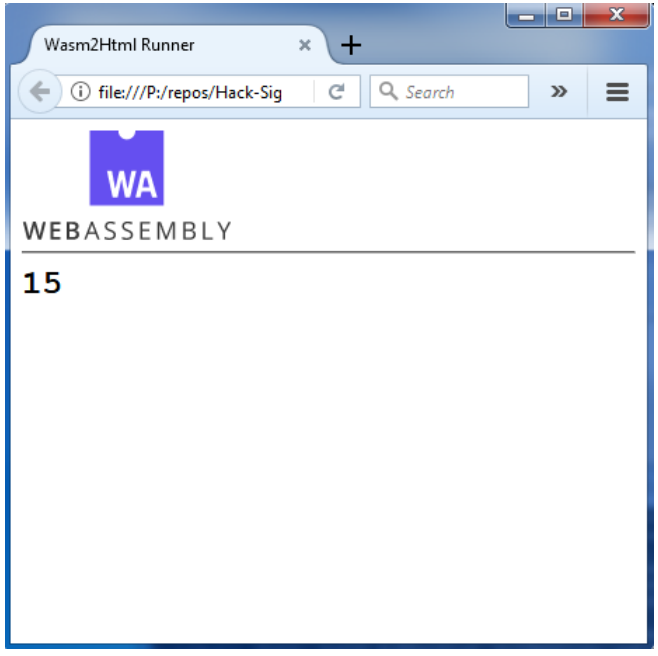
A complete program with imports, exports, and function calls:

```
(module

  (func $print (import "imports" "print") (param i32))

  ;; add function goes here

  (func $main

      i32.const 10
      i32.const 5
      call $add

      call $print

  )

  (export "main" (func $main))

)
```

Demo: ***Wasm2Html***

Simple little C# program that jams those bytes into an html file:

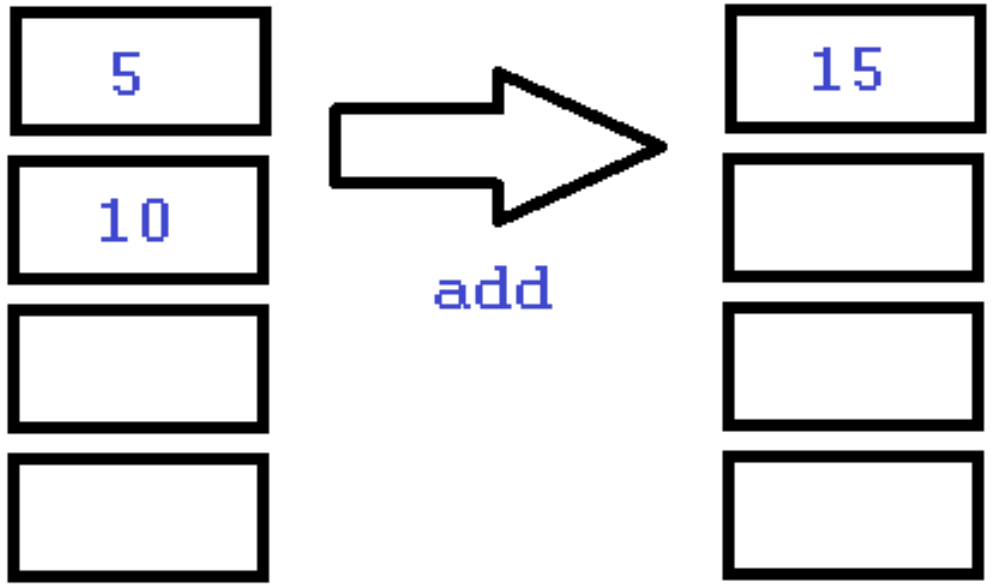
```
wat2wasm glue.wat -o out.wasm
wasm2html out.wasm
```



It's on GitHub

wasm is stack-based

Adding on the stack



If Construct

```
i32.const 9
i32.const 10
i32.eq
if
  i32.const 1
  call $print
else
  i32.const 0
  call $print
end
```

All comparison operators yield 32-bit integer results with 1 representing true and 0 representing false.

While Construct

```
block $block0
 loop $loop0          ;; while (true)
   get_local $n       ;; n=n+1
   i32.const 1
   i32.add
   set_local $n

   get_local $n       ;; if (n==10)
   i32.const 10
   i32.eq
   if
      br $block0      ;; break
   end

   get_local $n
   call $print

   br 0
 end
end
```

Building a Compiler with Irony

```
"Irony is a development kit for implementing languages
on .NET platform. Unlike most existing yacc/lex-style
solutions Irony does not employ any scanner or parser
code generation from grammar specifications written in
a specialized meta-language. In Irony the target language
grammar is coded directly in c# using operator overloading
to express grammar constructs."
```

In other words, Irony is an "internal" Domain Specific Language for creating General Purpose Languages.

Irony

http://irony.codeplex.com/

Roman Ivantsov AKA "Роман Иванцов"

Sounds Russian to me!

Let's collude!

But I needed to make some changes to the explorer because I think he was trying to interfere with my presentation.

All of this is on GitHub

Irony Demo

A simple text-based DSL with irony:

```
filters
[

   filter
   [
      from="blah blah";
      subject="blah";
   ]

   filter
   [
      from="blah blah";
      subject="blah";
   ]
]
```

Demo CovfefeScript (summing)

```
trigger main()
{
   cuck n
   cuck result

   n = 22 //253
   result = 0

   caterpillar
   {
      result = result +++ n

      shitpost(result)

      bong n

      if (n==0)
      {
         yerfired
      }
```

Demo CovfefeScript (fibonacci)

```
caterpillar
{
   bong n
   if (n==0)
   {
      yerfired
   }

   fib = n1 +++ n2
   n1 = n2
   n2 = fib

   shitpost(fib)
```

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

The End