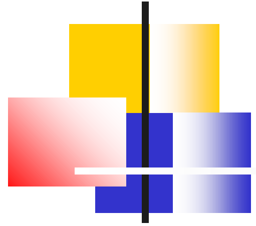
jeforth
Implementation Issues

SVFIG
C. H. Ting
February 27, 2021

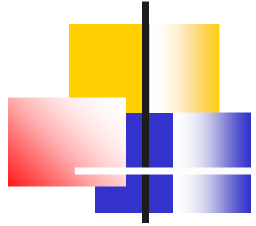
Jeforth6.13 Statistics

- **Jeforth613.js**
  - **Source code: 20,249 bytes**
  - **213 words**
  - **109 system words**
  - **18,454 bare byte**
- **8086eForth1.01**
  - **Source code: 47Kbytes**
  - **232 system words**



Irreducible Complexity

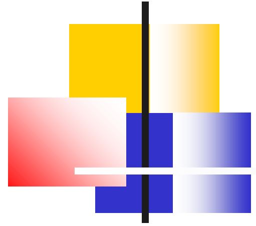
- “Things should be as simple as possible, but not simpler.” --Einstein.
- jeforth is at this ‘not simpler’ state.
- I like to call it ‘Irreducible Complexity.’



JavaScript

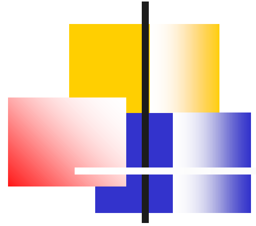
JavaScript is C with very flexible objects which can be anything:

```
var ip=0,wp=0,w=0;
var stack=[],rstack=[];
var tib="",ntib=0,base=10;
var idiom="";
var compiling=false;
var fence=0;
var newname;
var words[ … ];
```



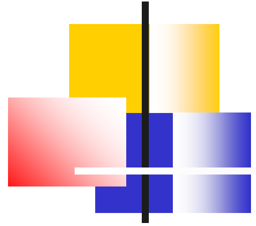
Terminology

- Words
  - Objects with name, code and data
- Dictionary
  - An array of all word objects
- Tokens
  - Index of word objects in a dictionary
- Idioms
  - Character strings in a source code file
  - May be words, numbers, or strings



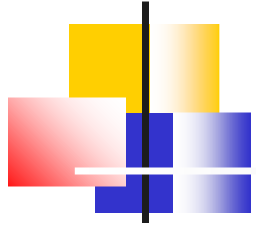
Word Objects

- 5 fields in a word object:
  - words[w].name
  - words[w].xt
  - words[w].pf, optional
  - words[w].qf , optional
  - words[w].immediate , optional



Defining Words

- Defining words
  - `: (colon)`
  - `constant`
  - `variable`
  - `create`
- Primitive words are JavaScript functions.
- Custom defining words are defined using `create-does` construct.



4 Types of Words

- **Primitive words**

```
{name:"parse" .xt:function(){idiom=parse();}}
```

- **Colon words**

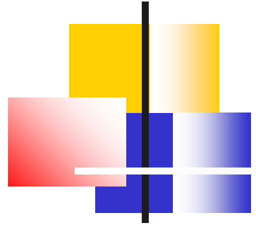
```
{name:"quit",xt:function(){nest();},pf:[1,2,3,0]}
```

- **Constants, Variables, Arrays**

```
{name:"a4" ,xt:function(){docon();},qf:[440  ]}
```

- **Words defined by defining words**

```
{name:"1/4",xt:function(){nest();},qf:[32]},
     pf[5,0,121,225,114,35,227,6]}
```



Interpreters

- **Outer or text interpreter**

```
: quit begin parse evaluate again
```

- **Inner interpreters**
  - **`nest()`** for colon words
  - **`docon()`** for constants
  - **`dovar()`** for variables and arrays
  - **`nest()`** for words defined by custom defining words
  - Code field functions for primitives words

Outer Interpreter

- **Outer or text interpreter**

```
: quit begin parse evaluate again
```

- `parse` **parses out next** `idiom` **in the input stream.**
- `evaluate` **evaluates the** `idiom` **and then execute it as:**
  - A word
  - A number
  - None of above

`nest()` Inner Interpreter

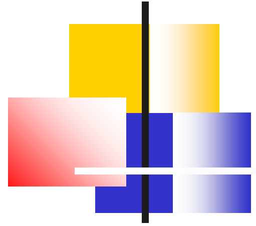
- **It is <u>the</u> inner interpreter**

```
function nest(){
rstack.push(wp);rstack.push(ip);
wp=w;ip=0;
while (ip>=0)
{w=words[wp].pf[ip++];words[w].xt();
  }
ip=rstack.pop();wp=rstack.pop();}

function exit(){ip=-1;}
```

- Primitive words
  - `xt:function(){   }`
- Colon words
  - `xt:nest(),pf:[   ]`
- Constants
  - `xt:docon(),qf:[   ]`
- Variables and arrays
  - `xt:dovar(),qf:[   ]`
- Words defined by defining words
  - `xt:nest(),pf:[   ],qf:[   ]`

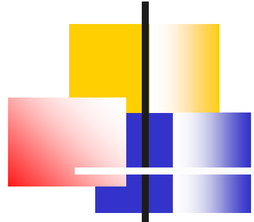
constant, variable, array

- **constant**
  - `xt:docon(),qf:[n]`
- **variable**
  - `xt:dovar(),qf:[0]`
- **create**
  - `xt:dovar(),qf:[ , , , , , , , , ]`
- **Words defined by defining words**
  - `xt:nest(),pf:[   ],qf:[ , , , , ]`



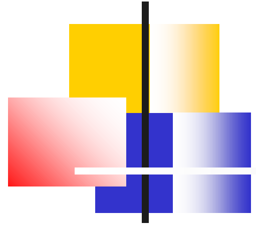
Stack Picture

```
negate xor or and mod / * - + pop
push r@ r> >r 2drop nip drop roll
pick 2over 2swap -rot rot swap 4dup
2over 2dup over dup branch evaluate
parse quit < >ok
< >ok
< >ok
1 2 3 4 5 6 7 < 1 2 3 4 5 6 7 >ok
< 1 2 3 4 5 6 7 >ok
< 1 2 3 4 5 6 7 >ok
```



Stack Words

```
{name:"dup”,xt:function(){stack=stack.concat(stack.slice(-1));}}
{name:"over”,xt:function(){stack=stack.concat(stack.slice(-2,-1));}}
{name:"2dup”,xt:function(){stack=stack.concat(stack.slice(-2));}}
{name:"2over”,xt:function(){stack=stack.concat(stack.slice(-4,-2));}}
{name:"4dup”,xt:function(){stack=stack.concat(stack.slice(-4));}}
{name:"swap”,xt:function(){stack=stack.concat(stack.splice(-2,1));}}
{name:"rot”,xt:function(){stack=stack.concat(stack.splice(-3,1));}}
{name:"-rot”,xt:function(){stack.splice(-2,0,stack.pop());}}
{name:"2swap”,xt:function(){stack=stack.concat(stack.splice(-4,2));}}
{name:"2over”,xt:function(){stack=stack.concat(stack.slice(-4,-2));}}
{name:"pick”,xt:function(){var j=stack.pop()+1;stack.push(stack.slice(-j,-j+1));}}
{name:"roll”,xt:function(){ var j=stack.pop()+1;stack.push(stack.splice(-j,1));}}
{name:"drop”,xt:function(){stack.pop();}}
{name:"nip”,xt:function(){stack[stack.length-2]=stack.pop();}}
{name:"2drop”,xt:function(){stack.pop();stack.pop();}}
```

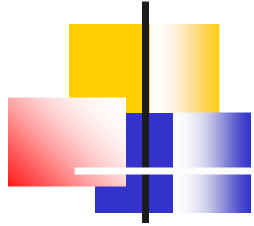


create allot ,

- **Arrays are defined by `create`.**
- **Arrays elements must be explicitly declared:**
  - `create xxx 1 , 2 , 3 ,`
  - `xt:dovar(),qf:[1,2,3]`
  - `create yyy 6 allot`
  - `xt:dovar(),qf:[0,0,0,0,0,0]`

`@ ! array@ array! q@`

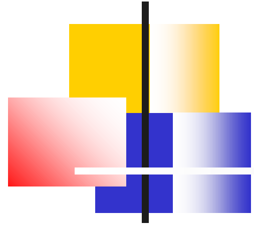
- `@` and `!` access variables, constants, and the first element of an array.
- `array@` and `array!` access an array by its token and an index into the array.
- Defining words access its array by `q@` and an index.



to a Constant

```
0 constant x 0 constant y
( libya is a prototype for drawing )
: proto 0 255 0 ;
: haiku 39996
  99 for r@ to x
    99 for r@ to y
      >r proto r@ image! r> 4 -
  next next drop show ;
haiku
```

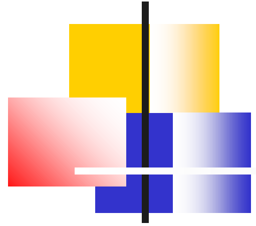
```
( sin is fun )
: switzerland   255  x 32 / sin 0.95 > y 32 / sin
  0.95 > or
   x 32 / sin 0.5 > and y 32 / sin 0.5 > and  255 *
  dup ;
' switzerland is proto haiku

: 4spire x 100 / 23 * sin y 100 / 1 swap - max
  x 100 / over / sin y 100 / 1 swap -  rot / sin
  2dup / sin 255 * rot 255 * rot 255 * rot ;
' 4spire is proto haiku
```



dump

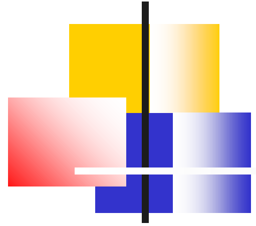
- It dumps the entire dictionary in a form that can be copied back to the source code file jeforth.js and make all the colon words available at boot-up.
- Kind of a turnkey system.

- **It decompiles a colon word.**
- **Tokens are substituted by their names.**
- **6 literal words are identified to display their literal values:**
  - **Number literals:** `dolit`
  - **Address literals:** `branch, 0branch, donext`
  - **String literals:** `dostr, dotstr.`



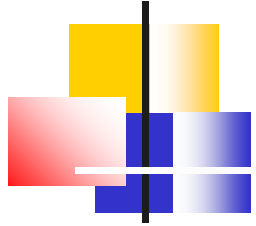
DO-LOOP

- for-nest **loops need only one primitive word** donext**.**
- DO-LOOP **requires a host of primitive words to support it.**
- **Converting** DO-LOOPs **to** for-next **loops:**
  - **Globally replace** LOOP **by** next
  - **Globally replace** 0 DO **by** 1 - for



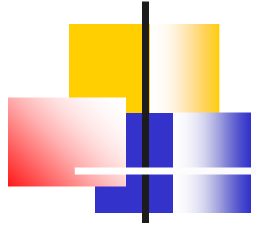
BLOCK

- Blocks are not essential to Forth.
- Blocks prevented the wide-spread use of Forth because they are not compatible with other operating systems.
- Text files are better media for Forth programming, data storage, and information sharing.



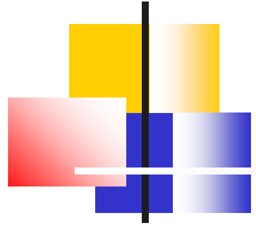
Audio Context

```
var AudioContext = window.AudioContext ||
  window.webkitAudioContext;
var audio = new AudioContext();
var amp = audio.createGain();
var osc1 = audio.createOscillator();
var osc2 = audio.createOscillator();
var osc3 = audio.createOscillator();
var osc4 = audio.createOscillator();
var osc5 = audio.createOscillator();
var osc6 = audio.createOscillator();
```



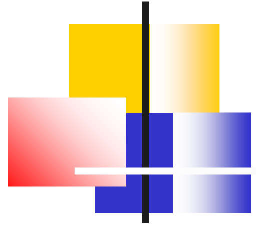
Conclusion

- JavaScript is very expressive.
- Sam Chen coded all eForth words in Javascript as primitives.
- All application words are compiled as colon words.
- `create-does` defines defining words.



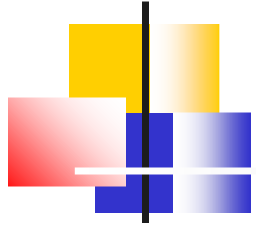
Conclusion

- **HTML allows jeforth603.html and jsBach603.html to design very friendly user interface for Forth.**
- **The input box accepts large text files and will be useful for you to program large applications.**



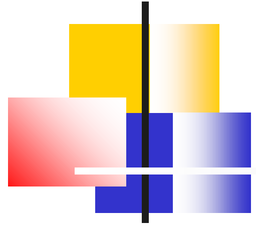
Conclusion

- Jeforth603.js system source code has 177 lines,18,454 bytes of text, and it defines 109 words.
- It is by far the smallest, simplest, and prettiest Forth I have ever produced.



Conclusion

- Lao Tze said:
- “For knowledge, learn one thing each day. For wisdom, forget one thing each day. When you have no-thing, you can do every-thing.”
- Have we forgotten things enough?



Google Drive Link

- **https://drive.google.com/file/d/15pjFff6ciNDIBR-jVlE_Eusob72YLPmO/view?usp=sharing**