A SLIGHTLY DIFFERENT FORTH COMPILER DESIGN

Joseph M. O'Connor

SVFIG Forth Day

Nov 2020

What I primarily use Forth for

- As a DSL embedded in a larger application
- I have written the following DSLs:
- (1) Creole Forth for Delphi/Lazarus
- (2) Creole Forth for Excel
- (3) Creole Forth for JavaScript
- (4) Creole Forth for Python
- (5) Creole Forth for Perl
- (6) Creole Forth for C# (still under development)
- My various Forth and other projects are available at https://github.com/tiluser

Development methodology

- (1) Import Creole Forth into the application
- (2) Set up a set of ‘primitives’ to handle the tasks necessary. This is a two-step process:
- A. Define a subroutine that follows a specified interfacing convention. It’s usually a method that takes a global properties structure as an argument and returns nothing.
- B. Add a reference to the dictionary with the BuildPrimitive method.
- (3) When enough primitives are available, start gathering into high-level definitions
- (4) It is possible to use multiple copies of Creole Forth for the same application, each with a different vocabulary set, which can be useful.

In some Forths, instructions can be defined in terms of assembly mnemonics. The primitives defined by Creole Forth play a similar role.

Compiler design

- No state variable.
- Compilation starts by pushing the IMMEDIATE vocabulary onto the vocabulary stack and ends with the popping of this vocabulary.
- Immediate words are simply words defined in the IMMEDIATE vocabulary, which the colon compiler will always search first.
- The interpreter is used by the compiler to handle the compilation.

Compilation steps

- (1) compileColon sets up a new CreoleWord definition. It populates the name field and fully qualified name field (name field + context vocabulary).
- (2) Using a loop, it does the following steps:
- 1. Looks up each word in the dictionary.
- 2. Builds a triplet with the following information : Name with context vocabulary, address, and compilation action.
- 3. Places the triplet in the PAD area. This is a simple list structure.
- 4, The PAD area contents are gone through one at a time and passed onto the interpreter.
- 5. Words with a COMPINPF action are compiled into the parameter field, while immediate words are executed. EXEC0 words (comments) are handled by moving the input stream pointer. Literals are handled by putting the literal-handling code before the literal.
- 6. doSemi pops the vocabulary stack and empties PAD.

Possible compilation actions

Name	Action
COMPINPF	Synonym for , (COMMA). Compiles an address into the parameter field
EXECUTE	Takes an address as a stack argument and executes it. Used to handle compiling words.
EXEC0	Moves the input stream pointer. Currently used for comments only.
LITERAL	Anything else that could not be found in the dictionary.

What PAD looks like during compilation : TEST1 IF HELLO ELSE TULIP THEN ;

Word	Token	Action	What compileColon does
IF	52	EXECUTE	Pushes 52 onto the stack, passes EXECUTE to the interpreter.
HELLO	5	COMPINPF	Pushes 5 onto the stack, passes COMPINPF to the interpreter.
ELSE	53	EXECUTE	Pushes 53 onto the stack, passes EXECUTE to the interpreter.
TULIP	6	COMPINPF	Pushes 6 onto the stack, passes COMPINPF to the interpreter.
THEN	54	EXECUTE	Pushes 54 onto the stack, passes EXECUTE to the interpreter.

Let's compile this to see what happens

: TEST1 IF HELLO ELSE TULIP THEN ;

Number	Word/Meaning	Vocabulary	Code Field	Help
55	0BRANCH	IMMEDIATE	Compiler.do0Branch	(flag --) Run-time code for IF
5	Index 0BRANCH jumps to if arg=0			
5	HELLO	FORTH	CorePrims.doHello	(--) Pops up an alert saying Hello World
56	JUMP	IMMEDIATE	Compiler.doJump	(--) Jumps unconditionally to the parameter field location next to it and is compiled by ELSE
7	Index JUMP jumps to			
57	doElse	IMMEDIATE	CorePrims.doNop	(--) Run-time code for ELSE
6	TULIP	FORTH	CorePrims.doTulip	(--) Pops up an alert saying Tulip
58	doThen	IMMEDIATE	CorePrims.doNop	(--) Run-time code for THEN

How compiling words work

- They're in the IMMEDIATE vocabulary.
- The IMMEDIATE vocabulary becomes available when the colon compiler pushes it onto the vocabulary stack and its visibility disappears when it's popped off.
- Therefore this vocabulary is always searched first – it won't be prevented from action by a word in another vocabulary with the same name.
- They're usually defined in terms of two primitives. The first has a compile-time action, and the second has a run-time action.
- Example:
- (1) IF is a primitive defined by compileIf.
- (2) It has code to look up the token of 0BRANCH in the dictionary and compile it into the parameter field during compilation.
- (3) 0BRANCH is a primitive defined by do0Branch and branches to the address in the following parameter field entry if false. If true it just increments the parameter field pointer to the next entry to execute.

Advantages to this approach

- Simplicity. The colon compiler can be defined in about 100 lines of code, depending on the base language,
- Eliminates the distinction between state-smart and state-dumb words.
- Code reuse. The compiler uses the interpreter to do its business

References

- Brodie, Leo. Thinking Forth. Copyright 1984, 1994.
- Smith, Norman E. Write Your Own Programming Language Using C++, 2nd Edition. Copyright 1996.
- Tracy, Martin, Anderson, Anita, and Advanced Micromotion, Inc. Mastering Forth. Copyright 1989.
- Ertl, M. Anton. State-smartness | Why it is Evil and How to Exorcise it

Questions?