

Design and Implementation of the Fifth Programming System

Benjamin Smith

April 5, 2002

Abstract

A description of the design of the fifth postfix applicative language, and of the compiler, including a commentary on its implementation and development

Contents

1	Introduction	2
1.1	Design Principles	2
1.2	An example	2
2	Getting Started	3
2.1	Syntax	3
2.2	The Simple Expression	3
2.3	Primitive Functions	3
2.3.1	Integer arithmetic	3
2.3.2	Integer I/O	4
2.3.3	Stack	4
2.4	Variables	4
2.4.1	Declaration using var	4
2.4.2	Recalling values	4
2.5	Comments	5
3	Language Reference	5
3.1	Tokens	5
3.2	Simple Expressions	5
3.3	Variables	6
3.3.1	var directive	6
3.3.2	Scope	6
3.4	Optimisation	6
4	Implementation	6
4.1	Internal representation	7
4.2	Passes	7
4.3	Tokenisation	7
4.4	Optimisation	7
4.4.1	Arithmetic reductions	7

4.5	Compilation	8
4.6	Register Allocation and Output	8
	Bibliography	8
	A Installation	8

1 Introduction

Fifth is an applicative¹ language, based on a postfix notation. It was first conceived as a language in a similar style to C to act as a low level structured alternative to assembly for device driver writing in an operating system project I was working on at the time. The primary goals were to have a language that allowed low level device access and predictable machine code output, while at the same time having many high-level features.

1.1 Design Principles

As previously mentioned the primary aim was to develop a language capable of low-level device access, but another consideration I had was that the language should be as clean and consistent as possible. As I had previously seen the forth language used for low-level programming, I settled upon a postfix syntax, and to be as clean and consistent as possible I decided that no prefix nor infix notation would be included. The primary limitation I saw with forth was that it had a weak type-checking system, and that many implementations were little more than fancy stack-based assemblers. I wanted a language that could have complex optimisations performed on it to produce efficient machine code making full use of all available registers. The highest-level language I had had any experience with was ML and so many of the features of fifth have been adapted from concepts found in ML.

1.2 An example

TODO: Include an example here

```
( vars.ft )
( test main program before a var )
5 4 + display ( output: 9 )

( test var )
2 3 + a var
a display ( output: 5 )

( the main program )
5 2 + ( 7 )
3 - ( 4 )
neg 5 swap - ( 9 )
dup display ( output: 9 )
```

¹The adjective applicative has been used in place of functional in this paper after jokes were made about the usability of this system

```
3 neg + ( 6 )
dup dup + + ( 18 )
( retval is 18 )
```

2 Getting Started

The aim of this chapter is to quickly introduce you to writing programs in fifth, before the more complicated formal grammar is explained. This is done so that some of the subtleties of the language can be more easily understood

2.1 Syntax

The input to the fifth system consists of *tokens*. A token is a series of non-whitespace characters terminated by either an EOF or a sequence of one or more whitespace characters. A token can either be a value to be pushed onto the stack, a function to transform items on the stack, or a directive. When the input is read it is first split into tokens and then scanned to find any directives present. Once a directive has been found any expressions that it requires as parameters are then scanned. This delayed process allows one to establish between the two different uses of a symbol (either as a variable recall or function call, or as a symbol to be pushed).

2.2 The Simple Expression

A simple expression is a set of tokens that processes an input stack of zero values to produce an output of exactly one value. For example the sum $2 + 3$ would be considered a simple expression and would be written `2 3 +` in fifth. When the last structure in a fifth program is a simple expression², the generated program will include an implicit output of the final value. This makes life easy to create programs performing simple sums and calculations. Before continuing you might like to try processing the above sum with fifth.

2.3 Primitive Functions

Fifth has a number of so-called primitive functions defined. At the machine level these are the units that are converted by the compiler into sequences of machine opcodes.

The primitives available currently include ones for integer i/o, integer arithmetic and simple stack manipulation.

2.3.1 Integer arithmetic

$i_1 i_2 + \rightarrow (i_1 + i_2)$

Adds the two parameters i_1 and i_2 pushing the single value on to the stack.

$i_1 i_2 - \rightarrow (i_1 - i_2)$

Subtracts i_2 from i_1 pushing the value on to the stack.

²This is in fact a simplification of the way the main parser works as the last expression is defined as being a complex expression taking no arguments and returning zero or one values.

i neg $\rightarrow (-i)$

Inverts the sign of i pushing the value on to the stack.

$i_1 i_2 *$ $\rightarrow (i_1 \times i_2)$

Multiplies i_1 and i_2 together pushing the value on to the stack.

$i_1 i_2$ div $\rightarrow (i_1 \div i_2) (i_1 \bmod i_2)$

Divides i_1 by i_2 pushing first the quotient then the remainder on to the stack.

2.3.2 Integer I/O

i display $\rightarrow \langle \text{empty} \rangle$

Prints i to the screen.

input $\rightarrow i$

Reads i from the input and pushes on to the stack.

2.3.3 Stack

$i_1 i_2$ swap $\rightarrow i_2 i_1$

Swaps i_1 and i_2 on the stack.

i dup $\rightarrow i i_{\text{copy}}$

Pushes a copy of the top element of the stack without consuming the element.

2.4 Variables

In fifth a variable is a way of naming a value so that it may be reused multiple times in the same program without (possibly expensive) re-evaluation occurring³.

Unlike in an imperative language a fifth variable is what many already familiar with programming would call a constant and not a memory cell in a von Neuman(sp?) machine. On the other hand mathematicians should recognise a fifth variable as having many of the same properties as an algebraic variable.

2.4.1 Declaration using var

The value for a variable is given in a **var** directive⁴.

$\langle \text{simpleexpression} \rangle \langle \text{symbol} \rangle \text{let}$

2.4.2 Recalling values

To use the value of a variable it is sufficient to mention its name and at compile time the appropriate instructions to retrieve its value will be generated. The system is smart enough to be able to determine the difference between a recall of a variable's value and a *symbol* token.

³In a lot of cases variables are only really needed to clarify the source code as the optimiser will locate and remove duplicate sub-expressions automatically.

⁴Please note that this is different from a **let** directive which is described later.

2.5 Comments

Comments in fifth follow the forth style of using parenthesis ‘(’ and ‘)’. To start a comment the opening parenthesis must appear on its own in a token (in most cases placing a space before the first character of the contents is sufficient), but no such restriction is placed upon the closing parenthesis, as the input is scanned but not tokenised until it is reached.

An example:

```
( this is a comment )
2 ( comments can go here ) 3 ( and here ) + ( and here )
( this is also a comment )
```

3 Language Reference

This section aims to offer the definitive reference (other than the source) to the fifth language, and specify its formal grammar.

3.1 Tokens

A token is defined as a sequence of one or more non-whitespace characters terminated by any sequence of one or more whitespace characters. In addition zero or more whitespace characters are allowed before the first token of a file.

Whitespace is defined as any of the space, new-line, carriage return, form feed or tab characters. All other characters (including symbols, digits and alphabetical characters) are considered to be non-whitespace.

In some cases what is allowed in a specific token might be more precisely specified but this restriction is implemented at the tokenisation stage when the role of a token is more precisely known.

This means that the string

```
 2 3 +
```

is parsed into three tokens: 2, 3, +. The quantity of white space appearing in the input has no bearing on the output of the tokenisation procedure.

As the tokens are read in they are quickly categorised into integers, symbols and directives, before being placed on the token list. An integer is recognised using the `strtol` standard library function. If this function consumes the whole of the token, the token is marked as an integer and the integer value stored with it on the token list. Otherwise the token is checked against the names of directives, and anything not an integer or directive is marked as a symbol.

3.2 Simple Expressions

A simple expression is a set of tokens that processes an input stack of zero values to produce an output of exactly one value. The start of an expression cannot be easily found without infinite lookahead, which is why expressions are parsed by reading the tokens list backward in a LIFO fashion. While the expression is being parsed a stack is maintained on which the return values of functions are stored.

The parsing procedure consists of two phases: a reverse traversal and a forward traversal. During the reverse traversal in addition to the data stack a counter is maintained of how many values are still to be seen. As each token is parsed this counter has the number of return values subtracted and the number of arguments added. When this counter drops to zero all the necessary tokens have been read and the reverse traversal is finished and the forward traversal is begun. At the start of parsing this counter is loaded with the value 1 as the expression should return 1 value. It is during the forward traversal that the main stack is used.

As an example suppose that the input consists of the following tokens:

1 2 3 4 + 5 +

When parsing starts the last token read was the final +, the stack counter is set to 1, and we are in the reverse phase. We first remove the top token from the stack, and look it up in the symbol table to find that it uses 2 arguments and gives 1 return value. We next adjust the value of the counter by $2 - 1$, to 2. The counter is then checked for equality with zero which returns false and so we continue to the next token.

TODO: finish commentary.

At the end we have parsed the expression indicated below.

1 2 3 4 + 5 +
simple expression

3.3 Variables

3.3.1 var directive

3.3.2 Scope

3.4 Optimisation

An implementation may perform any optimisation it desires as long as the order of execution of sideeffects is not affected. Therefore as long as the result of executing the optimised program is the same as executing the original the optimisation is valid.

It is strongly recommended that at a minimum arithmetic primitives that have constant arguments are replaced with constants themselves. This should also be extended to variables that are constant.

4 Implementation

Implementation of the compiler and the design of the language have mostly proceeded in parallel with new ideas in both being reflected in the other. This has allowed rapid checking of algorithms and grammars. During this phase of development efficiency wasn't a major concern, so the compiler was written in a multi-pass style with each pass separated into independant modules. The modules communicate together by processing a common internal representation of the program.

4.1 Internal representation

A modified form of triples [2, pages 254–255] were chosen for the internal representation of the source program. Triples were chosen over quadruples because of the unnecessary use of temporary variables. Triples also have the advantage of easily forming a tree structure. The triples used in this implementation have been modified to be held in a linked list negating the advantages of indirect triples, and also to have a variable number of arguments.

There are two ways of traversing this structure. Primarily all the triples are doubly linked together with previous and next pointers. Secondly the triples whose return values are the arguments to the triple are linked to, along with the triples who consume the triple's return values.

4.2 Passes

At the top level of the compiler there are four passes:

1. tokenisation and parsing of input files;
2. optimisation of program;
3. compilation of program to assembly;
4. register allocation and final output.

Each pass is self-contained and only communicates with the others through the symbol table and list of triples. Internally a pass may be constituted of multiple internal passes.

4.3 Tokenisation

TODO: copy from section above.

4.4 Optimisation

The optimiser is based upon the idea of reductions. Each type of triple has an associated set of reduction rules each consisting of a pattern to match and a set of operations to reduce the triple. Currently this is implemented by hand-coded C routines, but it is hoped that a system involving a dedicated reduction description format may be later employed.

The rest of this section will describe the reduction rules implemented by the optimiser.

4.4.1 Arithmetic reductions

The first reduction is the simplest, as it reduces an addition for which both parameters are constant to a constant which is their sum.

$$k i_1 + k i_2 \rightarrow k(i_1 + i_2)$$

4.5 Compilation

During the compilation phase the primitives are converted into assembly code. The assembly is stored in the same triples as the primitives were with code stored in auxillary data fields.

Each of the inputs and outputs of the primitive are matched up to registers or immediate parameters through the use of constraints. The constraint specifies which registers the operand is allowed to be stored in. In addition input constraints can also be marked as being the same as an output constraint, and this feature is used for instructions such as addition where the output is left in the same register as one of the inputs.

The string representing the instruction has embedded substitution sequences starting with a '%' which contains the number of the parameter whose register's name (or in the case of immediate operand its value) is to be substituted at that point in the instruction. A percentage sign can be obtained with a double percentage sign '%%'.

4.6 Register Allocation and Output

The register allocator is perhaps the most complicated system of algorithms in the compiler. It has the task of matching up return values and arguments to registers. Superficially this appears to be a simple task, easily managed with a bitmap of occupied registers but once constraints are introduced for instructions which can use only a subset of registers things become more complicated.

References

- [1] P. J. Brown. *Writing Interactive Compilers and Interpreters*. John Wiley & Sons, 1981.
- [2] David Gries. *Compiler Construction for Digital Computers*. Wiley International Editions. John Wiley & Sons, 1971.
- [3] F. R. A. Hopgood. *Compiling Techniques*. Macdonald/American Elsevier Computer Monographs. Macdonald & co., 1974.
- [4] Peter Wegner. *Programming Languages, Information Structures, and Machine Organisation*. McGraw-Hill, 1971.
- [5] Åke Wikström. *Functional Programming Using Standard ML*. Prentice Hall International Series in Computer Science. Prentice Hall, 1987.

A Installation

The current minimum requirements are having a GNU as compatible assembly for assembly of the compiler's output and a ANSI C compiler to compile the program. One freely available system for dos is djgpp⁵.

⁵<http://www.djgpp.org>

To compile the compiler it should only be necessary to type `'make'` as there are no external dependencies on libraries or tools. A full test suite is also included, to run it type `'make test'`.

To run the compiler simply use `'forth source.ft output.s'`. Typically the input has the extension `.ft` and the output `.s`. Then compile the assembly with the included `forthlib.c` file. Typically a command like `'gcc forthlib.c output.s'` would do this, putting the executable output into `a.out`.