

Introduction to StrongForth 3.1

Preface

This introduction to StrongForth has been written for those who already have collected some experience with Forth.

The basic idea behind StrongForth is the wish to add strong static type checking to a Forth system. Previous Forth systems and standards (including Forth 2012) were supposed to be *typeless* or *untyped*, which means they do not do any type checking at all. The interpreter and the compiler generally accept any word to be applied to the operands on the data and return stack. This behaviour grants total freedom to the programmer, but on the other side it is rather often a reason for type errors, which frequently cause system crashes and other more or less strange behaviour throughout the whole development phase.

StrongForth does not guarantee bug-free programs. It does not even grant the absence of crashes. But type errors will be greatly reduced. Furthermore, since interpreter and compiler know about the data types of the operands on the stack, they are able to choose the appropriate version of a word, if the dictionary contains several words with the same name, but different input parameter types. This is called operator overloading. As will be shown in this introduction, operator overloading allows a much more comfortable way of programming. Additionally, it is no longer necessary for you to invent individual names for words with the same semantics that are just applied to different data types.

Of course, strong static typing has some drawbacks, which might keep traditional Forth programmers from using it. First, it requires a higher degree of discipline, because the sources of all words having stack-effects have to be provided with precise stack diagrams. Second, interpreter and compiler will prohibit not only more or less dirty tricks, but sometimes also just *unusual* operations. For example, adding a flag to an address is not possible, although it might seem useful in some cases. And third, relying on a system that does all the type-checking itself, might lead to more careless programming.

The advantages and disadvantages of strong static type checking have already been discussed in the Forth community. The availability of StrongForth will certainly put more practical aspects into the previously rather theoretical discussion, allowing you to simply try it out by yourself.

First Steps

Let's begin with a few examples out of the first chapter of Leo Brodie's famous textbook *Starting Forth*:

```
15 spaces                                ok
```

When interpreting the number 15, the interpreter pushes this value on the stack and remembers that it is a single-cell unsigned integer number. `spaces` is a word that requires an integer number as the input parameter. Here's a possible definition of `spaces`:

```
: spaces ( unsigned -- )  
  0 ?do space loop ;
```

Well, this is not really exciting. At a first look, the only more or less interesting thing about it is the stack diagram. Standard Forth systems use `(n --)`, which is nothing but a comment. In StrongForth, it is interpreted source code, which compiles the stack diagram of `spaces` into the

dictionary. Additionally, it tells the compiler that the definition starts with an item of data type `unsigned` on the stack, and that it is expected to remove this item on exiting. Generally, each word in the dictionary includes full information about its stack effect.

So, let us now try a second example:

```
42 emit * ok
```

`emit` is a word that expects a number on the stack and displays the ASCII character associated with this number. We can also write

```
char * emit * ok
```

instead, because a character is some kind of a number. Even the following code works well:

```
char * . * ok
```

But wait ... Isn't `.` supposed to display a number, and not a character? Let's see:

```
42 . 42 ok
```

Yes, this still works. But how does `.` know whether it should print a number or an ASCII character? StrongForth actually provides more than one version of `.`. There are two version for displaying signed and unsigned numbers, and there's one version for displaying characters. The interpreter and the compiler take care of selecting the version that is suited best for the purpose. In this case, a number is displayed as a number, and a character is displayed as a character. When we write `42`, the interpreter pushes `42` onto the stack and keeps in mind that this is an unsigned number. When we write `char *`, the interpreter pushes exactly the same value onto the stack, but this time it takes a note that the item on top of the stack is a character. This note allows the interpreter to select the correct version of `.` `emit` doesn't make this difference. It displays each and every parameter as an ASCII character.

There are several other versions of `.` in StrongForth's dictionary. Just have a look at these:

```
3 4 = . false ok  
-16 . -16 ok
```

In this example, `=` takes the two items of data type `unsigned` and returns an item of data type `flag`. A dedicated version of `.` for flags delivers the appropriate result. The second example seems to be straight-forward, but it is not. Remember that `15`, `42`, `3` and `4` produced items of data type `unsigned`. `-16` produces an item of data type `signed`, and the interpreter finds a version of `.` suited for signed numbers. To enter a positive signed number, you have to precede it with a sign, for example `+16`. The advantage of distinguishing between signed and unsigned numeric literals becomes obvious when we try larger numbers:

```
4200000000 . 4200000000 ok  
+4200000000 . -94967296 ok
```

A standard 32-bit Forth system would always display `-94967296`, because it can not distinguish signed and unsigned numbers.

With the knowledge obtained so far, let's try out the compiler, still sticking to the examples in Leo Brodie's *Starting Forth*:

```

: star [char] * . ; ok
star * ok
cr
ok
cr star cr star cr star
*
*
* ok
: stars 0 do star loop ;
(do) ? undefined word
unsigned

```

Oops. What's that? `do` tried to compile `(do)`, which expects two numbers of the same data type on the stack, but there was only one. Thus, the compiler could not find an appropriate version of `(do)` in the dictionary, and throws an exception. Yes, we have to supply a stack diagram to `stars`:

```

: stars ( unsigned -- ) 0 do star loop ; ok
5 stars ***** ok
stars
stars ? undefined word

```

So, the compiler starts with an `unsigned` number on the stack, adds another one (`0`), and now `(do)` gets its input parameters. The last line just shows that `stars` will itself not be found in the dictionary, if the stack is empty.

Finally, let's complete Leo Brodie's example:

```

: margin cr 30 spaces ; ok
: blip margin star ; ok
: bar margin 5 stars ; ok
: f bar blip bar blip blip cr ; ok
f
*****
*
*****
*
*
ok

```

Data Types

In the previous section, we have introduced four data types: `unsigned`, `signed`, `character`, and `flag`. Actually, StrongForth knows a lot more data types, and it is even possible to define new, application-specific data types.

Data Type Structure

Having several different data types is certainly useful, but a large, unstructured quantity of data types would cause more problems than it solves. Since it should be possible to apply words like `dup` and `drop` to every data type, it would be necessary to supply separate versions of these words for each of them. Words with two input parameters, like `swap`, would have to be defined for each possible combination of two data types, which makes already 400 versions for 20 data types! `rot` would be even worse. And StrongForth provides far more than 20 different data types.

To solve this problem, StrongForth arranges all data types in a hierarchical structure. There are three data types at the root of this hierarchy: `single`, `double`, and `float`. All other data types

are direct or indirect subtypes of these three so-called *ancestor* data types. Here is an extract of StrongForth's data type structure:

```
single
  integer
    unsigned
    signed
  character
address
  caddress
  sfaddress
  dfaddress
logical
  flag
token
file
object
  stack-diagram
  input-stream
    terminal-input-stream
    string-input-stream
    file-input-stream
  output-stream
    terminal-output-stream
    string-output-stream
    file-output-stream
  control-flow
    origin
    destination
  exception-frame
  memory-space
  data-type-attributes
    class-attributes
  definition
    code-definition
    colon-definition
    created-definition
    single-definition
    double-definition
    float-definition
    deferred-definition
  vocabulary
double
  integer-double
  unsigned-double
  number-double
  signed-double
  data-type
float
```

Whenever the interpreter or the compiler tries to find a word in the dictionary, it accepts not only a word whose input parameters match the data types of the items on the stack *exactly*, but also a word whose input parameters are parents, grandparents or any other predecessors of those. Thus, only three versions of `dup` and `drop` cover all data types: one for `single`, one for `double`, and one for `float`. If, for example, the item on top of the stack has data type `unsigned`, `dup` for

single would match, because unsigned is a grandchild of single. Similarly, swap is overloaded nine times in order to deal with all combinations of single, double and float, including all direct and indirect subtypes, of both input parameters. rot even needs 27 overloaded versions. 27 seems to be quite a lot, but remember that only three of them, ROT, 2ROT and FROT, are specified in Forth 2012. Actually, rot is one of very few words in StrongForth having more than ten different overloaded versions.

Do you see the advantage of this concept? In order to swap two items on the stack, you just have to write swap, no matter whether those items were single-cell items, double-cell items, floating-point numbers or any combination of those. In Forth 2012, you have to write rot for swapping a single-cell and a double-cell item, or rot rot if you want to swap a double-cell and a single-cell item. That's not intuitive, is it?

Integers

Now, let's have a closer look at the data type structure. Some of the data types seem familiar to those explicitly specified in Forth 2012: unsigned is u, signed is n and character is char. These three data types are children of data type integer, which is itself is a child of single. integer is rarely used explicitly, but it is most useful as a common parent to the three data types. For example,

```
allot ( integer -- )
```

or

```
spaces ( integer -- )
```

can be applied to items of data types signed, unsigned and character, without having to define separate versions. But note that these two words may not be directly applied to addresses or flags, because that makes no sense. You might disagree, claiming that applying allot to a flag might be useful in certain applications. However, this would definitely be a programming trick, and it would be much clearer code writing

```
if -1 allot then
```

instead, although it is less efficient. If you want to keep this kind of efficiency in StrongForth, you'd have to use a type cast, which reveals the fact that it's a programming trick:

```
cast integer allot
```

Addresses

An address is not the same as an integer, because an address may not be added to another address. Two addresses may, however, be subtracted, giving an integer. There are several other restrictions regarding address arithmetic, like multiplication, but also some special features that only apply to addresses.

One of many Forth 2012 words returning an address is base:

```
decimal base @ . 10 ok
```

Okay, that works as expected. But how does @ know that the address on top of the stack is the address of an unsigned single-cell number? Obviously, the interpreter chooses the correct version of . to display an unsigned number. Let's try something else:

```
state @ . false ok
```

Same question: How does @ know ... ? The easiest way to get an answer is to get acquainted with StrongForth's version of .s:

```
45 -3 true char B .s unsigned signed flag character ok
```

Surprise, surprise! Instead of displaying the *data values*, `.s` shows the *data types* of the items on the stack. Well, what else did you expect from a strongly typed system? The information on data types is in many cases more useful than the actual numerical values. Now things are getting exciting:

```
abort \ clean stack
base .s caddress -> unsigned ok
```

What does that mean? `unsigned`, `flag`, `character` and so on are so-called *basic data types*. `caddress -> unsigned` is a *compound data type*, meaning *an address pointing to an unsigned character-size number*. Addresses have to be specific in the sense that addresses to different data types have to be distinguishable. `caddress`, which is a subtype of `address`, has the same meaning as `caddr` in Forth 2012: Whatever is stored at this address has the size of a character instead of the size of a cell. As you've seen, that is not necessarily a character. Anything that fits into 8 bits, like unsigned numbers between 0 and 255, signed numbers between -128 and +127, or flags, can be stored into a character-size memory location. Overloaded versions of `@` and `!`, that apply to addresses of data type `caddress`, have the semantics of `C@` and `C!` as specified by Forth 2012. The rest is easy to understand:

```
@ .s unsigned ok
. 10 ok
```

When `@` is supplied with an item of data type `caddress -> unsigned`, the interpreter finds an overloaded version which knows that it has to fetch an unsigned *character-size* number from memory and return it as `unsigned`. Other overloaded versions of `@` for data types `caddress -> signed` and `caddress -> flag` even perform proper sign extension.

Naturally, `caddress` is mostly used for dealing with character strings in memory. Here's an example of how to use `pad`, which actually returns the address of a character string buffer:

```
char F pad ! ok
char o pad 1+ ! ok
char r pad 2 + ! ok
char t pad 3 + ! ok
char h pad 4 + ! ok
pad .s caddress -> character ok
5 type Forth ok
```

Now, what about variables? A Forth 2012 variable can store anything from a signed number to an execution token. In StrongForth, the word `variable` has to be supplied with information about the data type that is supposed to be stored in it. This information can easily be provided by doing a small modification to the semantics of `variable`. In StrongForth, `variable` initializes the just created variable with the value of the item on top of the stack, while simultaneously taking over its data type:

```
char C variable x ok
x .s @ . address -> character C ok
```

An item to be stored into a variable must always have exactly the same data type as the one with which the variable had been initialized:

```
char D x ! ok
x @ . D ok
-13 x !
-13 x ! ? undefined word
signed address -> character
```

The error message means that the interpreter cannot find a word with the name ! that accepts the two input parameters `signed` and `data -> character`. Note that the second line of an error message always displays the data types of the items on the stack at the time the error was detected.

To show how powerful the concept of compound data types is, let's continue playing with variables:

```
x variable y ok
y .s address -> address -> character ok
@ .s address -> character ok
@ .s character ok
. D ok
```

Thus, a compound data type can consist of an arbitrary number of basic data types chained by `->`. It is therefore possible to store addresses of specific items in variables and generally operate with addresses of addresses of addresses and so on.

Logicals

An item of data type `logical` is a collection of individual bits in a single cell. Naturally, such items may not be involved in arithmetic operations like `+`, `-`, `*`, `/` or `negate`. On the other hand, logical operations like `and`, `or`, `xor` and `invert` may only be applied if the top item on the stack is of data type `logical`:

```
hex 12345678 55AA55AA or .
hex 12345678 55AA55AA or ? undefined word
unsigned unsigned
hex 12345678 55AA55AA cast logical or . 57BE57FA ok
```

`cast logical` is a so-called *type cast*, which converts an item of any data type into an item of data type `logical` without affecting its bit pattern. You can cast any data type to any other data type:

```
decimal ok
char X .s character ok
cast unsigned .s . unsigned 88 ok
```

Performing a logical operation on an integer is not necessarily dangerous, but you have to pay attention, because it is almost always a programming trick. StrongForth helps detecting programming tricks, because it requires type casts whenever you do something unusual. For example, one might decide to multiply a number by a power of 2 using `lshift`:

```
1000 6 lshift .
1000 6 lshift ? undefined word
unsigned unsigned
```

This does not work, because `lshift` is a logical operation that expects an item of data type `logical` on the stack:

```
1000 cast logical 6 lshift cast unsigned . 64000 ok
```

That looks rather awful. The code is certainly more readable with a pure arithmetic operation:

```
1000 64 * . 64000 ok
```

However, StrongForth's optimized compiler would actually compile a left shift instead of a multiply instruction if the number 64 is already known at compile time.

Data type `flag` is a subtype of `logical`. This ensures that all logical operations can directly be applied to flags, i. e., without the ugly type casts. A `flag` is just a `logical` with all bits set to the same value. The constants `true` and `false` both have data type `flag`:

```
true .s . flag true ok  
false .s . flag false ok
```

Tokens

Items of data type `token` are actually execution tokens, which are abbreviated with `xt` in Forth 2012. This data type is a direct subtype of `single`, in order to prevent any operations other than the few allowed ones to be applied to execution tokens.

Files

Data type `file` is used for file handles. It is directly derived from data type `single`, because this excludes arithmetic and logical operations to be applied to file handles. File handles are typically created with either `open` or `close`. Note that the names of these two words have been shortened with respect to Forth 2012. Thanks to the overloading mechanism in StrongForth, it is not necessary to give different Forth 2012 words like `create` and `create-file` unique names. This applies to other words from the *File-Access* word set as well, because most of them expect an input parameter of data type `file`, which ensures that the correct overloaded version is chosen by the interpreter.

Objects

StrongForth is object-oriented. One of the data types directly derived from data type `single` is `object`, which has in turn quite a number of subtypes. In contrast to simple data types like `integer`, `address`, `logical`, `token` and `file`, each object belongs to a class. Each class has a virtual method table, constructors and destructors. It may have class members as well as private and protected definitions that are only visible within the context of the definition of the class.

What is called a word in Forth 2012, is in StrongForth an object of class `definition`. The members of a definition are its name, a link to the previous definition, its stack diagram, and other attributes. You can obtain its execution token or a pointer to its data space, and compile or execute it. An item of data type `definition`, or one of its subtypes, will be produced by words like `'` and `:noname`. Other than in Forth 2012, `'` and `:noname` do not return execution tokens. The most important reason is that an execution token cannot directly be executed in StrongForth, because it bears no information about the stack diagram of the definition associated with it.

More information about this subject will be supplied in connection with a detailed explanation of `execute` in the StrongForth Reference manual. For now, the most interesting thing about `definition` is, that StrongForth provides another overloaded version of `.` for it. No, this is not the StrongForth synonym for `see`, but it displays the name and the complete stack diagram of a definition. Here are some examples:

```
' here . here ( -- address ) ok  
' name . name ( definition -- caddress -> character unsigned ) ok  
' >body . >body ( definition -- address ) ok
```

Input and output streams are objects as well. Instead of Forth 2012's complicated rules for selecting the input source, StrongForth just parses the default input stream, which may be the console, a character string, a file, a block or anything else. `emit` and `type` display characters on the default output stream. Dedicated words for writing to character strings, files and blocks are superfluous.

Class `vocabulary` covers Forth 2012 word lists. But a vocabulary can be more than a word list. It rather implements a description of how certain names are composed, for example, `integer` and `floating-point` numbers. You can define a new vocabulary that consults an external file or the Internet to find out whether a name is valid and how to compile it. You can define a vocabulary

that recognizes and compiles octal numbers, fixed-point decimal numbers or arbitrary-length numbers.

Other predefined classes are `stack-diagram`, `control-flow`, `exception-frame`, `memory-space` and `data-type-attributes`. And of course, you can define your own classes and create objects from them.

Double-cell Data Types

All members of the `double` branch of the data type structure occupy two cells in memory. This is not new to Forth. The big difference between StrongForth and Forth 2012 regarding double numbers is the fact, that Forth 2012 requires special names for those words that deal with double numbers, while StrongForth simply *overloads* the corresponding single number words. To duplicate two double numbers, one has to write `2DUP` in Forth 2012 and `dup` in StrongForth. Adding two double numbers is done with `D+` in Forth 2012 and `+` in StrongForth, as can be seen in this example:

```
10000000000000. dup + . 20000000000000 ok
```

`.` is overloaded as well. Overloading makes programming a lot easier. Actually, the complete StrongForth *Double-Number* word set consists of overloaded words. Since interpreter and compiler know about the data types of the items on the stack, they will always select the proper words.

In analogy to single numbers, StrongForth provides the predefined data types `integer-double`, `signed-double` and `unsigned-double`. The number `10000000000000.` in the above example is `unsigned-double`. When prefixed with a positive or negative sign, it will be interpreted as `signed-double`.

A new data type is `number-double`. It is only used between `<#` and `#>`, i. e., `<#` creates an item of this data type, while `#>` consumes it:

```
<# ( unsigned-double -- number-double )
#> ( number-double - caddress -> character unsigned )
```

This is an easy way to ensure that these two words are always paired. Since `#` and `#s` also work with items of data type `number-double`, syntax violations will immediately be detected by the compiler. As an example, here is a possible definition of `.` for signed double numbers:

```
: . ( signed-double -- )
  dup 0< swap abs <# #s swap sign #> type space ;
```

Note that the first four words being compiled are overloaded versions for double-cell numbers, and that `sign`, other than in Forth 2012, requires an item of data type `signed-double` as its input parameter.

Using a special data type for ensuring the proper syntax is a common technique in StrongForth. The subtypes of data type `control-flow`, which is a class and a subtype of `object`, are other examples. An object of data type `origin` is created by `if` and consumed by `then`. `begin` creates an object of data type `destination`, which is later consumed by `until` or `repeat`. `else` and `while` may be used in exactly the same way as specified in Forth 2012. `origin` and `destination` have themselves subtypes, which are not shown in the extract of the data type structure at the beginning of this section. Objects of these data types are used between `case` and `endcase`, and between `do/?do` and `loop/+loop`.

Another subtype of `double` is `data-type`. An item of data type `data-type` is, well, a data type. Words using a data type as input or output parameters are extensively used by the interpreter and the compiler. For example, specifying a data type within a stack diagram adds it to the stack diagram as an input or output parameter. An item of data type `data-type` is actually composed

of an identifier and a set of attributes. Data type attributes include information about whether a data type as part of a stack diagram references another data type and whether it is part of a compound data type. Data types can be created with `dt` in the same way as character literals are created with `char`. There's even an overloaded version of `.` for items of data type `data-type`:

```
dt signed-double .s . data-type signed-double ok
```

Stack Diagrams

When experimenting with displaying stack diagrams by using `.` for definitions, you might have found out that `'` always finds the most recent definition in the dictionary that matches the given name. Since many StrongForth words are overloaded, there often exist multiple occurrences of a name in the dictionary. This is a major difference to Forth 2012. You can use `words` for finding all overloaded versions of a name:

```
words .  
. ( float -- )  
. ( vocabulary -- )  
. ( definition -- )  
. ( data-type -- )  
. ( character -- )  
. ( flag -- )  
. ( signed -- )  
. ( single -- )  
. ( signed-double -- )  
. ( double -- ) ok
```

Trying `words` out with other names, you will almost certainly run into rather strange stack diagrams that look like these:

```
words dup  
dup ( float -- 1st 1st )  
dup ( double -- 1st 1st )  
dup ( single -- 1st 1st ) ok
```

Looking again at the data type structure, you'll find out that `1st` is not one of the predefined data types, neither is `2nd`, `3rd` and `th` in the following examples:

```
' >number . >number ( integer-double address -> character  
unsigned -- 1st 2nd 4 th ) ok  
words accept  
accept ( address -> character integer -- 3rd ) ok
```

These words obviously have a special meaning. Let's assume we define `xdup` as follows and try it out on an unsigned single number:

```
: xdup ( single -- single single ) dup ; ok  
4 xdup .s single single ok  
drop drop ok
```

Now we have two items of data type `single` on the stack instead of two items of data type `unsigned`. Trying, for example, to add those two items will fail, because `+` is only defined on data types `integer` and `address`, including subtypes, but not on data type `single`. That's why we have to use `1st` in the stack diagram of `dup`. When interpreting or compiling a word with `1st` as an output parameter, the data type of this parameter will be replaced with the data type of the first actual input parameter:

```

4 dup .s . . unsigned unsigned 4 4 ok
char j dup .s . . character character jj ok
base dup .s . . caddress -> unsigned caddress -> unsigned 4300813
4300813 ok

```

Now it works as expected. As can be seen in the last line of the example, `1st` also works correctly if the first input parameter has a compound data type. `2nd` and `3rd` work in a similar way, but reference the second or third basic data type in the input parameter list, respectively. To reference the fourth, fifth, sixth basic data type and so on, an unsigned number followed by `th` has to be used, as in the stack diagram of `>number`. This feature is perhaps one of most important keys to strong static typing in StrongForth. Many words use `1st`, `2nd`, `3rd` and `th` in their stack diagrams.

You might have noticed a small but important detail in the explanation of `2nd`, `3rd` and `th`. They do not reference the second (or third ...) *input parameter*, but the second (or third ...) *basic data type* in the input parameter list of a stack diagram. The necessity for making this difference becomes clear when having a closer look at the stack diagrams of `@`:

```

words @
@ ( dfaddress -> float -- 2nd )
@ ( sfaddress -> float -- 2nd )
@ ( caddress -> flag -- 2nd )
@ ( caddress -> signed -- 2nd )
@ ( caddress -> single -- 2nd )
@ ( address -> float -- 2nd )
@ ( address -> double -- 2nd )
@ ( address -> single -- 2nd ) ok

```

Let's only look at the last line of this list. Although `@` has only one input parameter, `2nd` references `single`, or, more generally, the tail of the compound data type standing for the first input parameter. Thus, when `@` is applied to the address of an unsigned single number, the data type of the output parameter is really that of an unsigned single number. As has been shown in the previous examples with variable `x` and variable `y`, it works as expected even if the tail of the referenced input parameter is itself a compound data type.

Another good example is `>number`, because this word has quite a lot of parameters:

```

>number ( integer-double caddress -> character unsigned -- 1st 2nd
4 th )

```

The first input parameter has the data type `integer-double`, the second one has the data type `caddress -> character` and the third one has the data type `unsigned`. Only the second input parameter has a compound data type. When the input parameter list is decomposed into basic data types, we get:

1. `integer-double`
2. `caddress`
3. `character`
4. `unsigned`

`1st` references the first basic data type, which is `integer-double` and nothing else. `2nd` references `caddress`. But since the basic data type `caddress` in this input parameter list is the head of a compound data type, `2nd` actually references the whole compound data type, namely `caddress -> character`. `3rd` would reference the third basic data type, `character`, which is the tail of the second input parameter. Finally, `4 th` references `unsigned`. `unsigned` is both the third input parameter and the fourth basic data type within the input parameter list.

Now it should be clear how several other words are defined. Have a look at the common arithmetic operators. As a general rule, the data type of the output parameter is the same as that of the first input parameter, thus allowing, for example, adding an integer to a character and still having a character on the stack afterwards. This should answer the question, why `+` is not defined as

```
+ ( integer integer -- integer ) \ wrong!
```

but as

```
+ ( integer integer -- 1st )
```

The most common application for data type references is in the output parameter list of stack diagrams. But data type references may also be used in the input parameter list, where they have a slightly different meaning. Look at the stack diagrams of the various overloaded versions of `!`:

```
words !
! ( float dfaddress -> 1st -- )
! ( float sfaddress -> 1st -- )
! ( single caddress -> 1st -- )
! ( float address -> 1st -- )
! ( double address -> 1st -- )
! ( single address -> 1st -- ) ok
```

Don't bother about what kind of data types `dfaddress` and `sfaddress` are. It's only the last line we shall investigate. `1st` means here, that the second input parameter is an address, which points to an item of exactly the same data type as the first input parameter. This is actually a restriction to the interpreter or compiler when trying to find a suitable version of `!` in the dictionary. It prevents you from trying to store something into a memory address that doesn't belong there. A simple example should clarify what this means:

```
char c variable x ok
char d x .s character address -> character ok
! ok
34 x .s unsigned address -> character ok
!
! ? undefined word
unsigned address -> character
```

The second `!` fails to match, because an unsigned single number may not be stored into a character variable.

Data Type Heaps

To keep track of the data types of the items on the stack, StrongForth has two data type heaps. Why two? Because StrongForth needs separate data type heaps for the interpreter and for the compiler.

The contents of the interpreter's data type heap can be displayed with `.s`. The items on the data type heap are mapped one to one to the items on the stack. If we have three items on the stack, we also have three data types on the data type heap, which can be either basic or compound data types.

The interpreter's data type heap is only used by the interpreter. There is no explicit type checking at runtime, because this would cause a tremendous performance penalty. That's the main difference between systems with static and dynamic type checking. Instead of doing dynamic type checking at runtime, StrongForth's compiler does static type checking at compile time. The compiler has its own data type heap, where it keeps the data types of the items that will be on the stack at runtime.

Since the interpreter is permanently present during compilation, having two separate data type heaps is a necessity. Immediate words generally use the interpreter data type heap, because they are

immediately executed. All other words are compiled, and use the compiler data type heap. Let's view an example:

```
: test 3 4 .s unsigned unsigned
+ .s unsigned
. .s
; ok
```

.s is an immediate word. In interpretation state, it displays the contents of the interpreter data type heap. In compilation state, it displays the contents of the compiler data type heap, as in this example. After having compiled two numeric literals, the compiler data type heap contains two times the data type `unsigned`. `+` is not immediate. The compiler finds a version of `+` that accepts two unsigned single numbers, and compiles it. It also updates the compiler data type heap by replacing the data types corresponding to the input parameters of `+` with the data type that corresponds to `+`'s output parameter, which is `unsigned`. `.` is also non-immediate. The compiler finds a version suitable for an unsigned single number and removes the data type of its input parameter from the compiler data type heap. Since `.` has no output parameters, the compiler data type heap is now left empty. `;` is immediate. Before compiling `exit`, it ensures that the contents of the compiler data type heap matches the assumed output parameter list of `test`. Both are empty, so everything is fine.

Here's a second example:

```
: counter ( unsigned -- ) 0 do i . loop ; ok
10 counter 0 1 2 3 4 5 6 7 8 9 ok
```

By default, a new definition is assumed to have no stack effect. This time, we have specified an explicit stack diagram. `)` initializes the compiler data type heap with one item of data type `unsigned`, so compilation starts with this item. Compiling `0` adds another `unsigned`, and `do`, an immediate word, consumes both by compiling `(do)`. `i` pushes `unsigned` on the data type heap, and `.` consumes it. `loop` checks that the contents of the compiler data type heap is the same as it was after `do` was executed, before compiling its own runtime semantics. Finally, `;` checks the congruence between the compiler data type heap and the output parameter list of `counter`.

That's what happens on the compiler data type heap. But what about the interpreter data type heap? We can easily watch it with `.s` by temporarily switching to interpretation state:

```
: counter [ .s ] colon-definition
( unsigned -- ) [ .s ] colon-definition
0 do [ .s ] colon-definition do-destination
i . loop [ .s ] colon-definition
; ok
```

`colon-definition`, which `:` pushes onto the interpreter data type heap, is the equivalent of what the Forth 2012 standard calls *colon-sys*. It identifies the current definition. `do` pushes another item onto the data stack and the interpreter data type heap, which is supposed to contain information for `loop` or `+loop`. `do-destination` is consumed by `loop`, and `;` consumes `colon-definition`. If we had tried to execute `;` before `loop`, the interpreter would not have found it in the dictionary, because `;` requires its input parameter `colon-definition` to be on top of the stack.

The Native-code Compiler

StrongForth's compiler creates native machine code instead of lists of tokens for a virtual machine. That makes the generated code fast. The code gets even faster, because the compiler adds some

sophisticated optimizations. And it becomes really fast because StrongForth does not even need a physical data stack.

A Forth without a data stack? Yes, that really works. StrongForth does not need a memory area used as the data stack, and there's no data stack pointer. All data you expect to be on the data stack are stored in the processor's general-purpose registers. But wait ... the underlying 32-bit x86 architecture has only six 32-bit registers: EAX, EBX, ECX, EDX, ESI and EDI. Doesn't that mean StrongForth's data stack is only six cells deep? No. The only restriction is that no word is allowed to have more than six cells of input parameters and six cells of output parameters. The only Forth 2012 words that fully use up these six cells are 2ROT and TIME&DATE. Do you consider a word expecting more than six cells on the stack or producing more than six cells consider useful? At least, it should be possible to factor it out in some way.

So, where does StrongForth store the data that do not fit into registers? Of course, they are pushed onto the return stack. The fact that these cells are inaccessible during the execution of a word, because the return address (*nest-sys* called in Forth 2012) is also stored on return stack, does not matter at all. Once the word currently being executed returns to its caller, the data stored on the return stack becomes available again. If the next word to be executed expects these data in certain registers, all required cells will be popped from the return stack.

The assignment between stack cells and registers may vary. As a consequence, words that only move or copy data on the stack do in most cases not compile any code at all. The compiler just changes the assignments. For example, let's assume the top cell of the data stack (TOS) is stored in register EAX and the next cell (NOS) is stored in register EBX. Now, if *swap* is being compiled, the compiler just reassigns the registers: EAX now contains NOS and EBX contains TOS. No machine code needs to be compiled. Here's an example:

```
: -rot ( single single single -- 3rd 1st 2nd ) rot rot ; ok
see -rot
code -rot ( ecx: single edx: single eax: single -- eax: 3rd ecx:
1st edx: 2nd )
00428973: ret,
endcode ok
```

The only machine code instruction compiled is *ret*,. In the stack diagram, the input and output parameters are preceded by the names of the registers they occupy. It's easy to see how *-rot* just changes the assignments in order to implement its semantics.

Let's view a more complex example:

```
: .byte ( single -- )
base @ hex swap s>d <# # # #> type base ! ;
see .byte
code .byte ( eax: single -- eax: ecx: edx: ebx: changed )
0042B49F: ecx 0041D00D byte[] movzx,
0042B4A6: 0041D00D byte[] 10 mov,
0042B4AD: edx edx xor,
0042B4AF: 00422B46 call, <#
0042B4B4: ecx push,
0042B4B5: 00422C4B call, #
0042B4BA: 00422C4B call, #
0042B4BF: 00422B4E call, #>
0042B4C4: 00406AAC call, type
0042B4C9: ecx pop,
0042B4CA: 0041D00D byte[] cl mov,
0042B4D0: ret,
endcode ok
```

.byte types the least significant byte of single in a two-digit hexadecimal format with no trailing space. base @ is compiled into only one machine code instruction. Remember that the variable base resides in a character-size memory location. swap compiles to nothing, s>d clears register edx, so the register pair eax/edx now contains a double-cell number.

<# # # #> type is compiled into a sequence of four subroutine calls. Register ecx, which holds the original value of base, needs to be saved to and restored from the return stack, because its contents is destroyed by # and #>. In the stack diagram of .byte, as it is displayed by see, you can see which registers are being destroyed by .byte itself. This information is included in the attributes of each definition, so that the compiler knows which registers need to be saved before it compiles the definition.

The last example in this section checks whether an item of data type flag assumes only the allowed values:

```
: ?bounds ( flag -- 1st )
dup if dup true <> if -289 throw then then ;
see ?bounds
code ?bounds ( ecx: flag -- ecx: 1st eax: changed )
0042B4D1: ecx ecx test,
0042B4D3: 0042B4E4 jz,
0042B4D5: ecx -01 cmp,
0042B4D8: 0042B4E4 jz,
0042B4DA: eax FFFFFFFD mov,
0042B4DF: 00407B4F call, throw
0042B4E4: ret,

endcode ok
```

The fact that the compiler knows the stack diagram of each word it compiles and which registers are used allows a number of optimizations not possible in other Forth systems. Replacing the data stack with the small set of general-purpose registers works fine, unless you need to handle more than six cells of input or output parameters.

When composing the stack diagram of a new word, StrongForth heuristically assigns registers to input parameters in such a way that these assignments most likely fit to what other words expect or return. Each data type has a default register, which is preferably used. E. g., the default register for addresses is ebx, for unsigned number and logicals including flags it is ecx. Double numbers are assigned to register pairs eax/edx, ebx/ecx or esi/edi.

A colon definition usually expects its parameters in specific registers, as can be seen in the above examples. However, the input parameters of some frequently used, precompiled words need not be in specific registers. Among these words are not only move and copy words like dup, drop and swap, but also @, ! and many arithmetical and logical operations. When compiling those words, the compiler saves explicit register shuffling in cases where the register assignments otherwise would not perfectly fit. Actually, this technique helps a lot in optimizing the generated code.

-rot as defined above would certainly be another candidate for such a technique. However, there's an even simpler way to accomplish the task:

```
: -rot ( -- ) postpone rot postpone rot ; immediate
```

This version covers all combinations of input parameters in one word, does not compile a subroutine call, and automatically uses all compiler optimizations StrongForth provides.