# Automatic Scoping of Local Variables

M. Anton Ertl

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
anton@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 4459
Fax.: (+43-1) 505 78 38

**Abstract.** In the process of lifting the restrictions on using locals in Forth, an interesting problem poses itself: What does it mean if a local is defined in a control structure? Where is the local visible? Since the user can create every possible control structure in ANS Forth, the answer is not as simple as it may seem. Ideally, the local is visible at a place if the control flow *must* pass through the definition of the local to reach this place. This paper discusses locals in general, the visibility problem, its solution, the consequences and the implementation as well as related programming style questions.

## 1 Introduction

Local variables can make Forth programming more enjoyable and Forth programs easier to read. Unfortunately, the local variables of ANS Forth [ANS93] are laden with restrictions, making them one of the weakest parts of the (otherwise excellent, within its scope) standard. They are restricted to be cell-sized and not addressable, they can only be declared outside control structures and they may interfere with using the return stack. The main contribution of the present paper is lifting the restrictions on where a local may be defined (see Section 4), but it also covers other aspects of the local variables in GNU Forth[1] (see Section 2).

## 2 Locals in GNU Forth

The starting point of GNU Forth's local variable syntax was TILE [Pat90]. Locals can be defined with

`{` *local*$_1$ *local*$_2$ ... `}`

or

`{` *local*$_1$ *local*$_2$ ... `--` ... `}`

E.g.,

```
: max { n1 n2 -- n3 }
 n1 n2 > if
   n1
 else
   n2
 endif ;
```

The similarity of locals definitions with stack comments is intended. A locals definition often replaces the stack comment of a word. The order of the locals corresponds to the order in a traditional stack comment[2] and everything after the `--` is really a comment.

This similarity has one disadvantage: It is too easy to confuse locals declarations with stack comments, causing bugs and making them hard to find. However, this problem can be avoided by appropriate coding conventions: Do not use both notations in the same program. If you do, they should be distinguished using additional means, e.g. by position.

TILE only supports cell-sized local values. We have extended the syntax to allow other data types: The name of the local may be preceded by a type specifier, e.g., `F:` for a floating point value:

```
: CX* { F: Ar F: Ai F: Br F: Bi -- Cr Ci }
\ complex multiplication
 Ar Br f* Ai Bi f* f-
 Ar Bi f* Ai Br f* f+ ;
```

GNU Forth currently supports cells (`W:`, `W^`), doubles (`D:`, `D^`), floats (`F:`, `F^`) and characters (`C:`, `C^`) in two flavours: a value-flavoured local (defined with `W:`, `D:` etc.) produces its value and can be changed with `TO`. A variable-flavoured local (defined with `W^` etc.) produces its address; it becomes invalid when the variable's scope is left. A local without type specifier is a `W:` local. Both flavours of locals are initialized with values from the data or FP stack.

I am also thinking about ways to define local variables with user-defined types, integrated into a more general wordset for supporting user data structures.

[1] The goal of the GNU Forth project is a portable, powerful and efficient implementation model for ANS Forth.

[2] [ANS93, Section A.13] presents a similar-looking syntax that has the order of locals reversed. Beware! [Hay92] shows how to get it right in ANS Forth.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth '94 Conference Proceedings, pages 31--37

| Name | Pronounciation | Stack effect | Purpose |
|---|---|---|---|
| `@local#` | fetch-local-number | →w | get a cell-sized local value |
| `f@local#` | f-fetch-local-number | →r | get an FP local value |
| `laddr#` | l-addr-number | →c-addr | get the address of a local |
| `lp+!#` | l-p-plus-store-number | → | increment (or decrement) the locals stack pointer |
| `lp!` | l-p-store | c-addr→ | set the locals stack pointer |
| `>l` | to-l | w→ | move a cell from the data to the locals stack |
| `f>l` | f-to-l | r→ | move a float from the float to the locals stack |

**Fig. 1.** Primitives for supporting the local variables (primitives with `#` take an inline argument)

### 2.1 Implementation

GNU Forth uses an extra locals stack. The most compelling reason for this is that the return stack is not float-aligned; using an extra stack also eliminates the problems and restrictions of using the return stack as locals stack. A few primitives allow an efficient implementation. In addition to the primitives shown in Fig. 1, specializations of these primitives for commonly occurring inline arguments are provided for efficiency reasons; combinations of conditional branches and `lp+!#` like `?branch-lp+!#` (the locals pointer is only changed if the branch is taken) are provided for efficiency and correctness.

A special area in the dictionary space is reserved for keeping the local variable names. `{` switches the dictionary pointer to this area and `}` switches it back and generates the locals initializing code. `W:` etc. are normal defining words. This special area is cleared at the start of every colon definition.

A special feature of GNU Forths dictionary is used to implement the definition of locals without type specifiers: every wordlist (aka vocabulary) has its own methods for searching etc. The original purpose of this feature is to make the mixing of different wordlist organizations possible, e.g. one wordlist is organized as a linked list, another uses a hash table. For the present purpose we defined a wordlist with a special search method: When it is searched for a word, it actually creates that word using `W:`. `{` changes the search order to first search the wordlist containing `}`, `W:` etc., and then the wordlist for defining locals without type specifiers.

### 2.2 GNU Forth and ANS Forth Locals

Of course GNU Forth also supports the ANS Forth locals wordset. The core of the word `(LOCAL)` is basically[3]:

```
nextname POSTPONE {  W: }
```

[3] `nextname ( addr u -- )` supplies the name for the next defining word.

This implementation is a little space-inefficient (due to alignment), but that will not hurt unless you are using these locals in a deeply recursive word. As a benefit, many of ANS Forth's restrictions do not apply to these locals. This makes porting programs to GNU Forth easy, but the other way might be problematic.

## 3 ANS Forth control structures

This section gives an overview of control structures in ANS Forth[4] for those who are not familiar with them. In contrast to, e.g., fig-Forth, ANS Forth permits and supports using control structures in a non-nested way. Information about incomplete control structures is stored on the control-flow stack. An *orig* entry represents an unresolved forward branch, a *dest* entry represents a backward branch target. A few words are the basis for building any control structure possible (see Fig. 2). These words are used to define the other standard control structure words [ANS93, Appendix A.3.2.2.2] (except `?DO`...`LOOP` etc., but the same principles apply there). They can also be used to build unusual control structures, e.g., a `BEGIN`...`UNTIL` with an additional exit:

| Word | Stack effect | Meaning |
|---|---|---|
| `IF` | →orig | conditional forward branch |
| `AHEAD` | →orig | unconditional forward branch |
| `THEN` | orig→ | target of forward branch |
| `BEGIN` | →dest | target of backward branch |
| `UNTIL` | dest→ | conditional backward branch |
| `AGAIN` | dest→ | unconditional backward branch |
| `CS-PICK` | ... n→... | control-flow stack manipulation |
| `CS-ROLL` | ... n→... | control-flow stack manipulation |

**Fig. 2.** Basic ANS Forth control structure words

[4] These ideas were developped (or documented, as he says) by Wil Baden.

```
BEGIN
  ...
IF [ 1 CS-ROLL ]
  ...
UNTIL
THEN
```

In this example the IF branches to the THEN, if it branches. Of course, programmers striving for readability will not use these techniques directly, but hide them in new control structure words.

## 4 Automatic Scoping

GNU Forth allows defining locals everywhere in a colon definition. After reading this section you may wonder if this feature is worth the trouble. It is: Good factoring requires that parts of definitions can be turned into separate colon definitions easily. If locals can only be defined outside control structures, factoring can become hard: If the part to be factored out references a local, this reference has to be first converted into stack manipulation words, then the factor can be introduced, and only then locals can be introduced into the factor. If locals are allowed anywhere, there is a good chance that the factoring can be performed without such changes, because the definitions of all referenced locals are included in the factor. Even if there have to be changes, they can be performed in small, easy steps; new locals can be defined at the right places before the factor is introduced. Another reason for this feature can be found in Section 5.1.

Removing the restriction that locals cannot be declared in control structures poses the following questions:

**Visibility** Where are such locals visible by name?
**Lifetime** How long do such locals live?

### 4.1 Visibility

**The Ideal:**Algol 60 and some of its offspring (e.g., C) solve the visibility problem by using explicit scope delimiters ({ and } in C), that are also used for building the control structures. However, explicit scoping cannot be transferred so easily to ANS Forth, where the language was not designed for this and the user can use control strucures in special ways and even define new control structures.[5]

Our ideal solution is to let a local be visible in all places that can only be reached through the definition of the local[6]. With normally structured programs this means that the locals are visible where you would expect it in block-structured languages, and sometimes a little longer (e.g., locals defined in a BEGIN...UNTIL are visible after the UNTIL). There are also the explicit scoping words SCOPE...ENDSCOPE.

The reasoning behind this solution is: We want to have the locals visible as long as it is meaningful (The user can always make the visibility shorter by using explicit scoping). In a place that can only be reached through the definition of a local, the meaning of a local name is clear. In other places it is not: How is the local initialized at the control flow path that does not contain the definition? Which local is meant, if the same name is defined twice in two independent control flow paths?

**... and Reality:** In principle the implementation of the visibility rules is very simple:

- At the start of a colon definition no locals are visible.
- A locals definition adds the local to the visible locals.
- After a control flow split (e.g., IF) the same locals are visible in both branches as before the split.
- After a control flow join (e.g., THEN) only those locals are visible that were visible in both of the joining branches.

Note that not all THENs etc. are control flow joins. E.g., if there is an EXIT before the THEN, the THEN can only be reached in one way and it is no control flow join.

Also, if an IF always goes in one direction, the corresponding THEN is no control flow join; however, since it is undecidable in general whether an IF always goes in one direction, the compiler always assumes that IFs and UNTILs can take both directions. This does not pose problems in practice: If the programmer knows that an IF must go in one direction, he can replace it with nothing or AHEAD (as appropriate), if he really wants the extended scopes; In the common cases of THROW and ABORT", he can use the word UNREACHABLE to tell the compiler that the control flow never reaches a place. If the programmer does not know that a conditional must go in one direction, he will not expect the compiler to make use of that fact anyway.

Apart from this fundamental problem, things are complicated a little by Forths one-pass compilation nature. In particular, BEGIN is a control flow join, but when it is compiled, the branch joining backwards is not known. All problems discussed in the

[5] Actually, this problem is also somewhat present in C: If there is a goto from outside a block into it, how are the local variables of the block initialized?

[6] In compiler construction terminology, all places dominated by the definition of the local.

following are due to this ignorance of the compiler. Perhaps the most insidious example is:

```
AHEAD
BEGIN
  x
[ 1 CS-ROLL ] THEN
  { x }
  ...
UNTIL
```

This should be legal according to the visibility rules. The use of `x` can only be reached through the definition; but that appears textually below the use.

From this example it is clear that the visibility rules cannot be fully implemented without major headaches. Our implementation treats common cases as advertised and the exceptions are treated in a safe way: The compiler makes a reasonable guess about the locals visible after a `BEGIN`; if it is too pessimistic, the user will get a false error about the local not being defined; if the compiler is too optimistic, it will notice this later and issue a warning. In the case above the compiler would complain about `x` being undefined at its use. You can see from the obscure examples in this section that it takes quite unusual control structures to get the compiler into trouble, and even then it will often do fine.

If the `BEGIN` is reachable from above, the most optimistic guess is that all locals visible before the `BEGIN` will also be visible after the `BEGIN`. This guess is valid for all loops that are entered only through the `BEGIN`, in particular, for normal `BEGIN`...`WHILE`...`REPEAT` and `BEGIN`...`UNTIL` loops and it is implemented in our compiler. When the branch to the `BEGIN` is finally generated by `AGAIN` or `UNTIL`, the compiler checks the guess and warns the user if it was too optimisitic:

```
IF
  { x }
BEGIN
  \ x ?
[ 1 cs-roll ] THEN
  ...
UNTIL
```

Here, `x` lives only until the `BEGIN`, but the compiler optimistically assumes that it lives until the `THEN`. It notices this difference when it compiles the `UNTIL` and issues a warning. The user can avoid the warning, and make sure that `x` is not used in the wrong area by using explicit scoping:

```
IF
  SCOPE
  { x }
  ENDSCOPE
BEGIN
[ 1 cs-roll ] THEN
  ...
UNTIL
```

Since the guess is optimistic, there will be no false error messages about undefined locals.

If the `BEGIN` is not reachable from above (i.e., after `AHEAD`, `AGAIN` or `EXIT`), the compiler cannot even make an optimistic guess, as the locals visible after the `BEGIN` may be defined later. However, our compiler assumes that the same variables are live as in the control flow edge represented by the top of the control flow stack. This covers the following case:

```
{ x }
AHEAD
BEGIN
  x
[ 1 CS-ROLL ] THEN
  ...
UNTIL
```

Other cases where the locals are defined before the `BEGIN` can be handled by inserting an appropriate `CS-ROLL` before the `BEGIN` (and changing the control-flow stack manipulation behind the `BEGIN`).

### 4.2 Lifetime

The right answer for the lifetime question would be: A local lives at least as long as it can be accessed. For a value-flavoured local this means: until the end of its visibility. However, a variable-flavoured local could be accessed through its address far beyond its visibility scope. Ultimately, this would mean that such locals would have to be garbage collected. Since this entails un-Forth-like implementation complexities, I adopted the same cowardly solution as some other languages (e.g., C): The local lives only as long as it is visible; afterwards its address is invalid (and programs that access it afterwards are erroneous).

### 4.3 Generated code

The lifetime rules support a stack discipline within a colon definition: The lifetime of a local is either nested with other locals lifetimes or it does not overlap them; if the definition of the local `b` can only be reached through the definition of `a` (i.e., `a` is visible at the definition point of `b`), certainly all places that can only be reached through the definition of `b` can only be reached through the definition of `a`; `a` is visible wherever `b` is visible, i.e., the scopes are nested.

At `BEGIN`, `IF`, and `AHEAD` no code for locals stack pointer manipulation is generated. Between control structure words locals definitions can push locals onto the locals stack. `AGAIN` is the simplest of the other three control flow words. It has to restore the locals stack depth of the corresponding `BEGIN` before branching. The code looks like this:

`lp+!#` *current-local-size − dest-locals-size*
`branch` ⟨begin⟩

`UNTIL` is a little more complicated: If it branches back, it must adjust the stack just like `AGAIN`. But if it falls through, the locals stack must not be changed. The compiler generates the following code:

`?branch-lp+!#` ⟨begin⟩ *current-l.-size − dest-l.-size*

The locals stack pointer is only adjusted if the branch is taken.

`THEN` can produce somewhat inefficient code:

`lp+!#` *current-local-size − orig-locals-size*
⟨orig target⟩:
`lp+!#` *orig-local-size − new-locals-size*

The second `lp+!#` adjusts the locals stack pointer from the level at the *orig* point to the level after the `THEN`. The first `lp+!#` adjusts the locals stack pointer from the current level to the level at the orig point, so the complete effect is an adjustment from the current level to the right level after the `THEN`.

The typical case where the inefficiency happens is when locals are defined between `IF` and `ELSE` in an `IF`...`ELSE`...`THEN` control structure. To avoid the inefficiency, `ELSE` could be changed to generate a `lp+!#` before the branch. A changed `THEN` would then patch the appropriate number into this instruction; then the `lp+!#` after the orig target would be unnecessary.

### 4.4 Implementation details

In a conventional Forth implementation a dest control-flow stack entry is just the target address and an orig entry is just the address to be patched. Our locals implementation adds a wordlist to every orig or dest item. It is the list of locals visible (or assumed visible) at the point described by the entry. Our implementation also adds a tag to differentiate between live and dead orig entries and other entries.

A few unusual operations have to be performed on locals wordlists:

`common-list ( list1 list2 -- list3 )` used in `THEN` to compute the list of locals visible after the `THEN`.

`sub-list? ( list1 list2 -- f )` used in `AGAIN` and `UNTIL` to check whether the assumption at the `BEGIN` was too optimistic.

`list-size ( list -- u )` computes the depth of the locals stack given the visible locals.

Several features of our locals wordlist implementation make these operations easy to implement: The locals wordlists are organised as linked lists; the tails of these lists are shared, if the lists contain some of the same locals; and the address of a name is greater than the address of the names behind it in the list.

Another important implementation detail is the variable `dead-code`. It is used by `BEGIN` and `THEN` to determine if they can be reached directly or only through the branch that they resolve. `dead-code` is set by `AHEAD`, `EXIT` etc., and cleared at the start of a colon definition, by `BEGIN` and usually by `THEN`.

Counted loops (`?DO`...`LOOP` etc.) are similar to other loops in most respects, but `LEAVE` requires special attention: It performs basically the same service as `AHEAD`, but it does not create a control-flow stack entry. Therefore the information has to be stored elsewhere; traditionally, the information was stored in the target fields of the branches created by the `LEAVE`s, by organizing these fields into a linked list. Unfortunately, this clever trick does not provide enough space for storing our extended control flow information. Therefore, we introduce another stack, the leave stack. It contains the control-flow stack entries for all unresolved `LEAVE`s.

Local names are kept until the end of the colon definition, even if they are no longer visible in any control-flow path. In a few cases this may lead to increased space needs for the locals name area, but usually less than reclaiming this space would cost in code size.

## 5 Programming Style

The freedom to define locals anywhere has the potential to change programming styles dramatically. In particular, the need to use the return stack for intermediate storage vanishes. Moreover, all stack manipulations (except `PICK`s and `ROLL`s with run-time determined arguments) can be eliminated: If the stack items are in the wrong order, just write a locals definition for all of them; then write the items in the order you want.

This seems a little far-fetched and eliminating stack manipulations is unlikely to become a conscious programming objective. Still, the number of stack manipulations will be reduced dramatically if local variables are used liberally (e.g., compare `max` in Section 2 with a traditional implementation of `max`).

This shows one potential benefit of locals: making Forth programs more readable. Of course, this benefit will only be realized if the programmers continue to honour the principles of factoring instead of using the added latitude to make the words longer.

### 5.1 Is `TO` necessary?

Not for our locals. In fact, it should be avoided. Without `TO`, every value-flavoured local has only a single assignment and many advantages of functional languages apply to Forth. I.e., programs are easier to analyse, to optimize and to read: It is clear from the definition what the local stands for, it does not turn into something different later.

`TO` can of course be avoided by turning a value-flavoured local into a variable-flavoured local, but this approach does not yield the benefits. Instead, the right approach is to define additional value-flavoured locals. E.g., with the ANS Forth restriction of not defining locals inside control structures an implementation of `strcmp` might look like this:

```
: strcmp { addr1 u1 addr2 u2 -- n }
 u1 u2 min 0
 ?do
   addr1 c@ addr2 c@ - ?dup
   if
     unloop exit
   then
   addr1 char+ TO addr1
   addr2 char+ TO addr2
 loop
 u1 u2 - ;
```

Here, `TO` is used to update `addr1` and `addr2` at every loop iteration. `strcmp` is a typical example of the readability problems of using `TO`. When you start reading `strcmp`, you think that `addr1` refers to the start of the string. Only near the end of the loop you realize that it is something else.

If locals can be defined in control structures, this can be avoided by defining two locals at the start of the loop that are initialized with the right value for the current iteration.

```
: strcmp { addr1 u1 addr2 u2 -- n }
 addr1 addr2
 u1 u2 min 0
 ?do { s1 s2 }
   s1 c@ s2 c@ - ?dup
   if
     unloop exit
   then
   s1 char+ s2 char+
 loop
 2drop
 u1 u2 - ;
```

Here it is clear from the start that `s1` has a different value in every loop iteration.

## 6 Performance

To understand the performance impact of using our local variables in an interpretive system, I compared the versions of `max` and `strcmp` with versions that do not use local variables (see Fig. 3). The comparison was performed on a 486DX2/66[7] using indirect threading and without the TOS optimization under Linux. The times are given in $\mu$s for one invocation of the code sequences `1 max` and `2dup 2dup strcmp drop` respectively. The string comparison input data is a string of 17 characters. In addition to the times with and without locals the time using the primitives `max` and `compare` is given.

| | with locals | without locals | primitive | w./w.o. ratio |
|---|---|---|---|---|
| `max` | 3.56 | 2.69 | 0.85 | 1.32 |
| `strcmp` | 83.20 | 70.50 | 3.90 | 1.18 |

**Fig. 3.** Timing results (times in $\mu$s/invocation)

The slowdown factor of using locals is due to the execution of more primitives (e.g., 14 instead of 12 per character in `strcmp`). Originally there was also a large overhead due to fetching inline arguments, resulting in slowdowns of 1.58 for `max` and 1.41 for `strcmp`. This overhead has been eliminated mostly by using versions of the primitives specialized for frequent inline arguments (e.g., `8lp+!` as specialization of `lp+!#` with the inline argument 8).

The slowdown is not caused by automatic scoping, as can be seen from the `max` example which does not use automatic scoping. The overhead should be acceptable in most cases. The improved productivity of using locals should make up for the increased amount of optimizing the running program if optimization is necessary. Besides, if you really want speed, then using an interpretive Forth implementation with its factor 5-10 of overhead is a bad idea anyway. A good native code implementation will allocate value-flavoured locals into registers, just like stack items. I.e., in a good native code implementation using locals is as fast as using stack manipulation words.

## 7 Related Work

The need for locals has been realized by the Forth community for a long time. This has led to a large

[7] Due to poor register allocation GNU Forth is significantly slower on this processor than Forths written in assembly. However, this should not have a significant influence on the results of these measurements.

variety of implementations and to a number of papers ([Tev89] cites seven papers on locals).

[Gla86] uses a fancy syntax for definitions using locals, which allows defining locals only for the whole definition, i.e., not within control structures. The implementation uses an extra stack. He reports 20%-30% run-time penalty (consistent with our results for a similar implementation) and a factor 2 speedup in programming and debugging time when using locals. A remarkable feature of this system is that in a defining word the equivalent of the `DOES>`-part can access locals of the `CREATE`-part.

[Tev89] keeps the locals on the data stack. This mechanism can be understood best by thinking not about local variables but by giving names to data stack locations. The names can be declared anywhere. The scope of a name ends when the item is taken from the stack.[8] Apart from consuming the items implicitly, there are also means to explicitly drop down to a named item. The implementation needs to keep track of the data stack depth, so it contains everything that is needed for a stack checker [Hof93]. The purpose of keeping the locals on the data stack is to avoid the overhead of moving them to a different stack. This works quite well, for some words there is even a speedup if locals are used.

TILE [Pat90] also keeps the locals on the data stack, but accesses the locals through a frame pointer, avoiding the need to keep track of the stack depth. Locals can be declared only outside control structures. They are automatically removed from the locals stack upon exiting the colon definition and everything further up (i.e., return values) is moved down.

## 8 Conclusion

Local variables have been a hot topic in Forth for a long time and ANS Forth will not be the last word on it. The main contribution of the present paper is that locals can be defined anywhere in a colon definition; their visibility scopes are determined automatically: A local is visible in all parts of the colon definition that can be reached only through the definition of the local. Such a powerful locals facility leads to a programming style that uses fewer stack manipulation words and is more readable. The use of `TO` with locals can be avoided, making the programs more functional and easier to analyse. Our locals implementation in a threaded code system makes words using locals about 1.2–1.3 times slower than equivalent words that do not use locals, but the increased readability and reduced programming time gained by using locals is worth this overhead.

[8] Certainly the lifetime ends there. The visibility should end there, too, but the listing included in the paper does not implement this.


## Acknowledgements

The anonymous referees, Bernd Paysan, Lennart Benschop, Andi Krall, Franz Puntigam and Manfred Brockhaus provided valuable comments on draft versions of this paper.


## References


[ANS93] *Draft proposed American National Standard — Forth (X3J14 dpANS6)*, 1993.

[Gla86] Harvey Glass. Towards a more writable Forth syntax. In *Dr. Dobb's Toolbook of Forth*, chapter 21, pages 169–181. M&T Books, Redwood City, CA 94063, 1986. Reprinted from *Proceedings of the 1983 Rochester Forth Applications Conference*.

[Hay92] John R. Hayes. User-defined local variable syntax with ANS Forth. *SigForth Newsletter*, 4(2), 1992.

[Hof93] Ulrich Hoffmann. Static stack effect analysis. In *EuroFORTH '93 conference proceedings*, Mariánské Lázně (Marienbad), 1993.

[Pat90] Mikael Patel. *TILE Release 2.1*, 1990. Available via ftp from any GNU archive site.

[Tev89] Adin Tevet. Symbolic stack addressing. *Journal of Forth Application and Research*, 5(3):365–379, 1989.