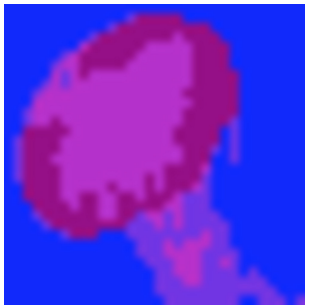
Hacking Vim script

kat0h

About me



Taken at an aquarium in Tronto

- Name: Kota Kato
- Position: undergraduate (The University of Electro-Communications)
- Github / Twitter: kat0h
- Interest: Programming Languages

https://github.com/kat0h/my-clisp

https://github.com/kat0h/kawk

https://gist.github.com/kat0h/ab0078de8c004c30bea757dfccca38df

Contribute

Commits on Jan 19, 2023

patch 9.0.1218: completion includes functions that don't work
kat0h authored and brammool committed on Jan 19, 2023

Commits on Jan 18, 2023

patch 9.0.1212: cannot read back what setcellwidths() has done
kat0h authored and brammool committed on Jan 18, 2023

Commits on Aug 17, 2022

patch 9.0.0219: cannot make a funcref with "s:func" in a def function
kat0h authored and brammool committed on Aug 17, 2022

Commits on Aug 5, 2022

patch 9.0.0140: execute() does not use the "legacy" command modifier
kat0h authored and brammool committed on Aug 5, 2022

Preparation

How to get Vim's source code

- The Vim source code is maintained on GitHub.
  - background (cf. slideshare)
- https://github.com/vim/vim
  - Search google for "vim vim"
- git clone https://github.com/vim/vim
  - `cd vim; make -j; sudo make install`
  - From this point on, the discussion assumes that you are at the repository root.

make tags using ctags

- Vim's source code has **many** conditional macros, making features like LSP definition jumps difficult to work correctly
  - HEAVY CPU load
- Make functions or definitions index using Ctags
  - After creation, you can jump to the definition using `<C-]>` or :tag. (cf. `:h usr_29`)
- ~~$ ctags -R~~ make tags (thx! k.takata)
  - make tags file for all files.

How to debug Vim script

Add a function to set a break point

- Implement internal Vim script function to set gdb break points.
- Q: Where is the definition of Vim script function
  - A: `global_functions` in `src/evalfunc.c`
    - array of `funcentry_T` struct
    - `:tag funcentry_T`

Add function into function list

- Implement function
- Add the definition to `global_functions` (the function list).
  - Pay attention to the order.
    - If you make a mistake → it will appear in completion but cause an error.
    - (Completion uses linear search; execution uses binary search.)
      - cf. `evalfunc.c` `find_internal_func_opt()`

Implement function

- Define a no-op function `f_debug()`
  - Add the function and prototype declaration in `src/evalfunc.c`

function prototype

```
static void f_debug(typval_T *argvars, typval_T *rettv);
```

Implementation

```
static void
f_debug(typval_T *argvars, typval_T *rettv)
{
    return;
}
```

Implement function

- `argvars`
  - Function arguments.
- `rettv`
  - Function return value (passed as an argument, it's the number `0`).
- `behavior`
  - When called without arguments, it returns `0`.

Implementation

```
    static void
f_debug(typval_T *argvars, typval_T *rettv)
{
    return;
}
```

Implement function

- `argvars`
  - Function argum
- `rettv`
  - Function retur ent, it's the number `0`).
- `behavior`
  - When called w 0`.

?

Implementation

```
3958     static vo
3959 f_debug(typval_T *argvars, typval_T *rettv)
3960 {
3961     return;
3962 }
```

```
struct typval_S
{
    vartype_T   v_type;
    char        v_lock;     // see below: VAR_LOCKED, VAR_FIXED
    union
    {
        varnumber_T     v_number;       // number value
        float_T         v_float;        // floating point number value
        char_u          *v_string;      // string value (can be NULL)
        list_T          *v_list;        // list value (can be NULL)
        dict_T          *v_dict;        // dict value (can be NULL)
        partial_T       *v_partial;     // closure: function with args
#ifdef FEAT_JOB_CHANNEL
        job_T           *v_job;         // job value (can be NULL)
        channel_T       *v_channel;     // channel value (can be NULL)
#endif
        blob_T          *v_blob;        // blob value (can be NULL)
        instr_T         *v_instr;       // instructions to execute
        class_T         *v_class;       // class value (can be NULL)
        object_T        *v_object;      // object value (can be NULL)
        typealias_T     *v_typealias;   // user-defined type name
    }           vval;
};
```

type of value

Is the value locked? (see also :h :lockvar)

src/structs.h

Implement function

- For example, to make `debug()` return the string `"Vim conf"`.
  - Set `v_type` to `VAR_STRING`.
  - Set the string in `vval.v_string` using `vim_strsave()`

Implementation

```
    static void
f_debug(typval_T *argvars, typval_T *rettv)
{
    rettv->v_type = VAR_STRING;
    rettv->vval.v_string = vim_strsave("Vim conf");
    return;
}
```

Implement function

- Add the definition of the debug function
  - With no arguments and no method calls
  - It returns a string

```
1752 static funcentry_T global_functions[] =
1753 {
1754     {"abs",                1, 1, FEARG_1,      arg1_float_or_nr,
1755 +--198 行: ret_any,        f_abs},
1953                            ret_number,         f_cursor},
1954     {"debug",              0, 0, 0, NULL, ret_string, f_debug},
1955     {"debugbreak",         1, 1, FEARG_1,      arg1_number,
1956                            ret_number,
1957 #ifdef MSWIN
1958             f_debugbreak
```

Description of `funcentry_T` type

- The definition of the built-in function is written.
  - Name, number of arguments, and implementation.
  - Type information for Vim9 script.

```
typedef struct {
    char  *f_name;
    char  f_min_argc;
    char  f_max_argc;
    char  f_argtype;
    argcheck_T  *f_argcheck;
    type_T  *(*f_retfunc)(int argcount, type2_T *argtypes, type_T **decl_type);
    void  (*f_func)(typval_T *args, typval_T *rvar);
} funcentry_T;
```

Implement function

- `f_name`
  - name of function

```
typedef struct
{
    char        *f_name;        // function name
    char        f_min_argc;     // minimal number of arguments
    char        f_max_argc;     // maximal number of arguments
    char        f_argtype;      // for method: FEARG_ values; bits FE_
    argcheck_T  *f_argcheck;    // list of functions to check argument types;
                                // use "arg_any" (not NULL) to accept an
                                // argument of any type
    type_T      *(*f_retfunc)(int argcount, type2_T *argtypes,
                                                    type_T **decl_type);
                                // return type function
    void        (*f_func)(typval_T *args, typval_T *rvar);
                                // implementation of function
} funcentry_T;
```

Implement function

- `f_min_argc`
  - minimal number of arguments

```
typedef struct
{
    char        *f_name;        // function name
    char        f_min_argc;     // minimal number of arguments
    char        f_max_argc;     // maximal number of arguments
    char        f_argtype;      // for method: FEARG_ values; bits FE_
    argcheck_T  *f_argcheck;    // list of functions to check argument types;
                                // use "arg_any" (not NULL) to accept an
                                // argument of any type
    type_T      *(*f_retfunc)(int argcount, type2_T *argtypes,
                                                    type_T **decl_type);
                                // return type function
    void        (*f_func)(typval_T *args, typval_T *rvar);
                                // implementation of function
} funcentry_T;
```

Implement function

- `f_max_argc`
  - maximal number of arguments

```
typedef struct
{
    char        *f_name;        // function name
    char        f_min_argc;     // minimal number of arguments
    char        f_max_argc;     // maximal number of arguments
    char        f_argtype;      // for method: FEARG_ values; bits FE_
    argcheck_T  *f_argcheck;    // list of functions to check argument types;
                                // use "arg_any" (not NULL) to accept an
                                // argument of any type
    type_T      *(*f_retfunc)(int argcount, type2_T *argtypes,
                                                    type_T **decl_type);
                                // return type function
    void        (*f_func)(typval_T *args, typval_T *rvar);
                                // implementation of function
} funcentry_T;
```

Implement function

- `f_argtype` (for Vim9 script)
  - In the case of a method call (`v->func()`), `v` becomes the **first argument** by default.
  - Select from around line 1709 of `evalfunc.c`, or choose `0`.

```
typedef struct
{
    char        *f_name;        // function name
    char        f_min_argc;     // minimal number of arguments
    char        f_max_argc;     // maximal number of arguments
    char        f_argtype;      // for method: FEARG_ values; bits FE_
    argcheck_T  *f_argcheck;    // list of functions to check argument types;
                                // use "arg_any" (not NULL) to accept an
                                // argument of any type
    type_T      *(*f_retfunc)(int argcount, type2_T *argtypes,
                                                    type_T **decl_type);
                                // return type function
    void        (*f_func)(typval_T *args, typval_T *rvar);
                                // implementation of function
} funcentry_T;
```

Implement function

- FEARG_1
  - `(-1)->abs()` ⇔ `abs(-1)`
- FEARG_2
  - `(1)->printf("value: %d")` ⇔ `printf(“value: %d”, 1)`

Implement function

- `f_argcheck` (for Vim9 script)
  - A list of the number and types of arguments
  - Select the appropriate item from the list around line 1090 of `evalfunc.c`

```
typedef struct
{
    char        *f_name;        // function name
    char        f_min_argc;     // minimal number of arguments
    char        f_max_argc;     // maximal number of arguments
    char        f_argtype;      // for method: FEARG_ values; bits FE_
    argcheck_T  *f_argcheck;    // list of functions to check argument types;
                                // use "arg_any" (not NULL) to accept an
                                // argument of any type
    type_T      *(*f_retfunc)(int argcount, type2_T *argtypes,
                                                    type_T **decl_type);
                                // return type function
    void        (*f_func)(typval_T *args, typval_T *rvar);
                                // implementation of function
} funcentry_T;
```

Implement function

```
1086 /*
1087  * Lists of functions that check the argument types of a builtin function.
1088  */
1089 static argcheck_T arg1_blob[] = {arg_blob};
1090 static argcheck_T arg1_bool[] = {arg_bool};
1091 static argcheck_T arg1_buffer[] = {arg_buffer};
1092 static argcheck_T arg1_buffer_or_dict_any[] = {arg_buffer_or_dict_any};
1093 static argcheck_T arg1_chan_or_job[] = {arg_chan_or_job};
1094 static argcheck_T arg1_dict_any[] = {arg_dict_any};
1095 static argcheck_T arg1_dict_or_string[] = {arg_dict_any_or_string};
1096 static argcheck_T arg1_float_or_nr[] = {arg_float_or_nr};
1097 static argcheck_T arg1_job[] = {arg_job};
1098 static argcheck_T arg1_list_any[] = {arg_list_any};
1099 static argcheck_T arg1_list_number[] = {arg_list_number};
1100 static argcheck_T arg1_string_or_list_or_blob_mod[] = {arg_string_list_or_blob_mod};
1101 static argcheck_T arg1_list_or_dict[] = {arg_list_or_dict};
1102 static argcheck_T arg1_list_string[] = {arg_list_string};
1103 static argcheck_T arg1_string_or_list_or_dict[] = {arg_string_or_list_or_dict};
1104 static argcheck_T arg1_lnum[] = {arg_lnum};
1105 static argcheck_T arg1_number[] = {arg_number};
1106 static argcheck_T arg1_string[] = {arg_string};
1107 static argcheck_T arg1_string_or_list_any[] = {arg_string_or_list_any};
1108 static argcheck_T arg1_string_or_list_string[] = {arg_string_or_list_string};
1109 static argcheck_T arg1_string_or_nr[] = {arg_string_or_nr};
1110 static argcheck_T arg2_any_buffer[] = {arg_any, arg_buffer};
1111 static argcheck_T arg2_buffer_any[] = {arg_buffer, arg_any};
1112 static argcheck_T arg2_buffer_bool[] = {arg_buffer, arg_bool};
1113 static argcheck_T arg2_buffer_list_any[] = {arg_buffer, arg_list_any};
1114 static argcheck_T arg2_buffer_lnum[] = {arg_buffer, arg_lnum};
1115 static argcheck_T arg2_buffer_number[] = {arg_buffer, arg_number};
1116 static argcheck_T arg2_buffer_string[] = {arg_buffer, arg_string};
1117 static argcheck_T arg2_chan_or_job_dict[] = {arg_chan_or_job, arg_dict_any};
1118 static argcheck_T arg2_chan_or_job_string[] = {arg_chan_or_job, arg_string};
1119 static argcheck_T arg2_dict_any_list_any[] = {arg_dict_any, arg_list_any};
1120 static argcheck_T arg2_dict_any_string_or_nr[] = {arg_dict_any, arg_string_or_nr};
```

Implement function

- `f_retfunc` (for Vim9 script)
  - A function that returns the type of the return value.
  - Select the appropriate one from around line 1237 of `evalfunc.c`

```
typedef struct
{
    char        *f_name;        // function name
    char        f_min_argc;     // minimal number of arguments
    char        f_max_argc;     // maximal number of arguments
    char        f_argtype;      // for method: FEARG_ values; bits FE_
    argcheck_T  *f_argcheck;    // list of functions to check argument types;
                                // use "arg_any" (not NULL) to accept an
                                // argument of any type
    type_T      *(*f_retfunc)(int argcount, type2_T *argtypes,
                                                    type_T **decl_type);
                                // return type function
    void        (*f_func)(typval_T *args, typval_T *rvar);
                                // implementation of function
} funcentry_T;
```

Implement function

- `f_func`
  - implementation of function

```
typedef struct
{
    char        *f_name;        // function name
    char        f_min_argc;     // minimal number of arguments
    char        f_max_argc;     // maximal number of arguments
    char        f_argtype;      // for method: FEARG_ values; bits FE_
    argcheck_T  *f_argcheck;    // list of functions to check argument types;
                                // use "arg_any" (not NULL) to accept an
                                // argument of any type
    type_T      *(*f_retfunc)(int argcount, type2_T *argtypes,
                                                    type_T **decl_type);
                                // return type function
    void        (*f_func)(typval_T *args, typval_T *rvar);
                                // implementation of function
} funcentry_T;
```

Implement function

- Try executing the implemented `debug()` function.

```
static void
f_debug(typval_T *argvars, typval_T *rettv)
{
    rettv->v_type = VAR_STRING;
    rettv->vval.v_string = vim_strsave("Vim conf");
    return;
}
```

```
~
~
~
:echo debug()
```

```
~
~
~
Vim conf
```

Implement function

```
function! Func()
  while 1
    call debug()
  endwhile
endfunction

call Func()

~
~
:so %
```

```
    0[||                    4.6%]   4[|||                   5.2%]   8[||                    1.3%] 12[|||
    1[||                    2.6%]   5[|                     0.7%]   9[|                     0.7%] 13[||||
    2[||||||||||||||||||||100.0%]   6[||                    2.0%]  10[|                     0.6%] 14[||
    3[                      0.0%]   7[|                     0.7%]  11[||                    1.3%] 15[|
  Mem[||||||||||||||||                                 3.86G/31.0G] Tasks: 121, 718 thr, 230 kthr; 2 running
  Swp[||                                                217M/40.0G] Load average: 0.66 0.78 0.94
                                                                    Uptime: 2 days, 12:07:43

  Main   I/O
    PID USER      PRI  NI  VIRT   RES   SHR S  CPU% MEM%   TIME+  Command△
 117016 kat0h      20   0 35796 16688 15024 R  99.3  0.1  0:18.52 ./src/vim -u NONE
 117017 kat0h      20   0 35796 16688 15024 S   0.0  0.1  0:00.00 ./src/vim -u NONE
```

Use termdebug to debug Vim

- For debugging Vim, **Termdebug** is useful.
  - Termdebug is a built-in GDB frontend for Vim.
    - You can use the mouse (!!)
  - (It likely) doesn't work in Neovim.

```
[0] 0:vim*                                                                  [10/22(火) 20:11]
(gdb)
Thread 1 "vim" hit Breakpoint 1, ex_echo (eap=0x7fffffffcc80)
    at eval.c:7321
7321    {
```

GDB's Prompt

```
                          VIM - Vi IMproved

                           version 9.1.757
                        by Bram Moolenaar 他.
              Vim はオープンソースであり自由に配布可能です

                   Vimの登録ユーザーになってください!
              詳細な情報は            :help register<Enter>

              終了するには            :q<Enter>
              オンラインヘルプは      :help<Enter> か  <F1>
              バージョン情報は        :help version9<Enter>

:echo "Hello, Vimconf 2024"
gdb [実行中]                    1,1          全て  debugged-program [アクティブ]          1,0-1          全て
 Step   Next   Finish   Cont   Stop   Eval
        }
    vim_free(tofree);
}

/*
 * ":echo expr1 ..."    print each argument separated with a space, add a
 *                      newline at the end.
 * ":echon expr1 ..."   print each argument plain.
 */
    void
ex_echo(exarg_T *eap)
01{
    char_u      *arg = eap->arg;
    typval_T    rettv;
    char_u      *arg_start;
    int         needclr = TRUE;
    int         atstart = TRUE;
    int         did_emsg_before = did_emsg;
    int         called_emsg_before = called_emsg;
    evalarg_T   evalarg;

    fill_evalarg_from_eap(&evalarg, eap, eap->skip);
src/eval.c                                                          7321,1          94%
```

debugee program

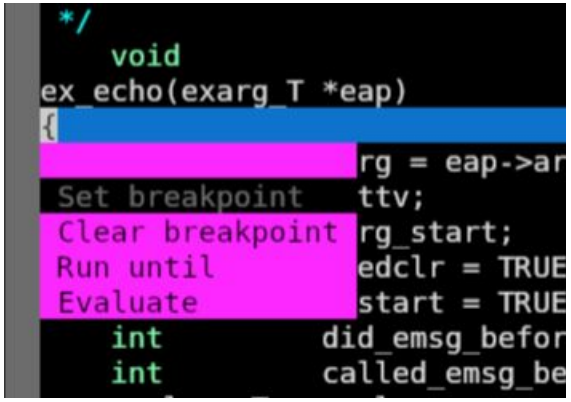
Source code
(you can set breakpoints with mouse!)

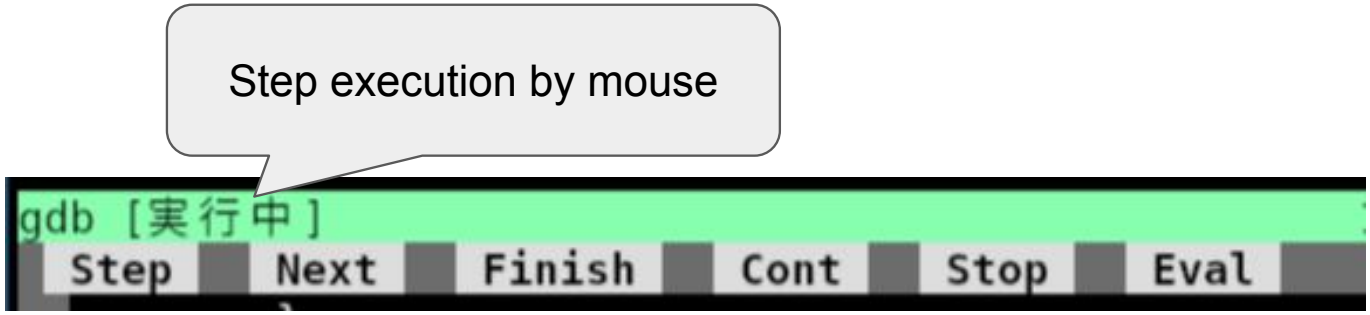
Use termdebug to debug Vim

- Prepare a debug-enabled version of Vim.
  - Edit src/Makefile
    - `CFLAGS = -g`
    - `STRIP = /bin/true`
  - `./configure –prefix=$PWD/debug_build`
  - `make -j && make install`
  - `file ./debug_build/bin/vim`
    - with debug_info, not stripped

Use termdebug to debug Vim

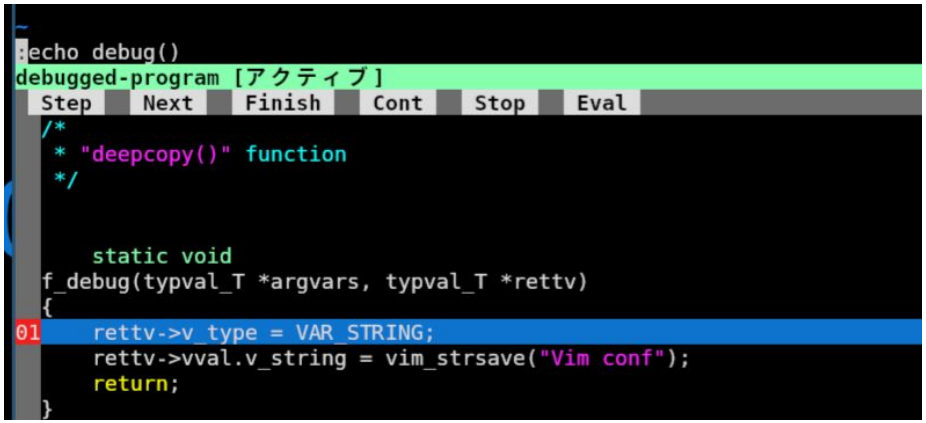
- `:packadd termdebug`
- `:Termdebug ./debug_build/bin/vim`
  - At the GDB prompt, use `run -u NONE`
- Open the source code in a regular window.
  - Right-click on the code to set a breakpoint.





Use termdebug to debug Vim

- `(gdb) b f_debug()`
  - Set a breakpoint on the `f_debug` function.
  - When you execute the `debug()` function at the timing you want to observe behavior, Vim will pause execution.



Check when the code was changed

- When you want to understand the intent of the code, `git blame` is useful.
- Why was the `funcentry_T` entry increased?
  - `$ git blame src/evalfunc.c`
  - **94738d8fab** appears very frequently.
    - It seems likely that changes related to Vim9 script are involved.

```
                                        1753 static funcentry_T global_functions[] =
                                        1754 {
4 years ago   Bram Moolenaar  94738d8fab 1755     {"abs",                1, 1, FEARG_1,            arg1_float_or_nr,
2 years ago   Bram Moolenaar  73e28dcc61 1756                         ret_any,            f_abs},
3 years ago   Yegappan Laksh  7237cab8f1 1757     {"acos",                1, 1, FEARG_1,            arg1_float_or_nr,
2 years ago   Bram Moolenaar  73e28dcc61 1758                         ret_float,            f_acos},
2 years ago   Bram Moolenaar  fa1039760e 1759     {"add",                2, 2, FEARG_1,            arg2_listblobmod_item,
4 years ago   Bram Moolenaar  94738d8fab 1760                         ret_first_arg,            f_add},
3 years ago   Yegappan Laksh  7237cab8f1 1761     {"and",                2, 2, FEARG_1,            arg2_number,
4 years ago   Bram Moolenaar  94738d8fab 1762                         ret_number,            f_and},
3 years ago   Yegappan Laksh  1a71d31bf3 1763     {"append",                2, 2, FEARG_2,            arg2_setline,
3 years ago   Bram Moolenaar  3af15ab788 1764                         ret_number_bool,    f_append},
```

git blame tips

- Vim includes `.git-blame-ignore-revs`
  - This is used to specify commits to ignore in `git blame`.
  - cf. https://github.com/vim/vim/blob/master/.git-blame-ignore-revs
- lambdalisue/vim-gina shows readable git blame

Check when the code was changed

- Use `git bisect` to identify when the `funcentry_T` type changed.
- `$ git bisect start HEAD fc73515f7`
  - `git bisect good`
  - `git bisect bad`
- patch 8.2.0149: Maintaining a Vim9 branch separately is more work

Execution flow of Vim script

Command execution

- `:echo "hello"` and the `execute()` function
  - Executed in the `do_cmdline()` function in `ex_docmd.c`
  - Each command is processed by the `do_one_cmd()` function.
  - **The structure is similar to that of a shell script**
- In Vim script, each command is parsed and executed every time.
  - from a performance perspective, this mechanism is quite inefficet

Expression evaluation

- The functions `eval1` through `eval9` **parse and execute** expression strings step by step.
- The associativity of operators can sometimes change depending on the values.
  - cf. https://thinca.hatenablog.com/entry/20131127/1385487671
  - Constructing an AST can, in the worst case, require memory space on the order of 2^n

Expression evaluation

eval1: `e ? e : e` (ternary operator), `e ?? e` (falsy operator)
eval2: `e || e` (logical OR)
eval3: `e && e` (logical AND)
eval4: `==`, `=~`, `!=`, `!~`, `>`, `>=`, `<`, `<=`, `is`, `isnot` (comparison operators)
eval5: `<<`, `>>` (bitwise shifts)
eval6: `+`, `-` (arithmetic operators), `.`, `..` (string concatenation)
eval7: `*`, `/`, `%` (arithmetic operators)
eval8: Type casting (specific to Vim9 script)
eval9: Top-level expressions like numbers, functions, variables, and unary operators

Evaluation Variable

- Vim script has **`g:b:w:t:s:l:v:`** scopes.
  - every scope has different hashmaps
  - `eval_variable()` find and return the variables

Function evaluation

- `eval_func()`
  - in `eval9()` (top level expressions)
  - Functions declared with the `:function` command are stored in memory as plain strings.
  - `call_func()` execute it

Garbage Collection

- The **reference counting** memory management system cannot free circular references.
- ```
  :let l = [1, 2, 3]
  :let d = {9: l}
  :let l[1] = d
  ```
-
  - This drawback can be resolved by periodically inspecting unreachable objects.
  ```

Execution flow of Vim9 script (bonus)

What is Vim9 script

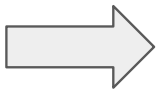
- A programming language faster than Vim script
  - Implemented in Vim
- Programs are compiled into stack machine bytecode before execution.
  - Defined using the `:def` command, using `:vim9cmd` command and using `:vim9script` command the top of source file
  - cf: `:h vim9`

What is the stack VM

- Vim9 script is executed on a stack-based virtual machine (VM).
  - Converting code into the VM's bytecode.
- **push a → push 2 → add**
- Use the `:disassemble` command to inspect the internal representation.
  - Test function for the main implementation.

```
def! Func()
  var a = 1
  echo a + 2
enddef

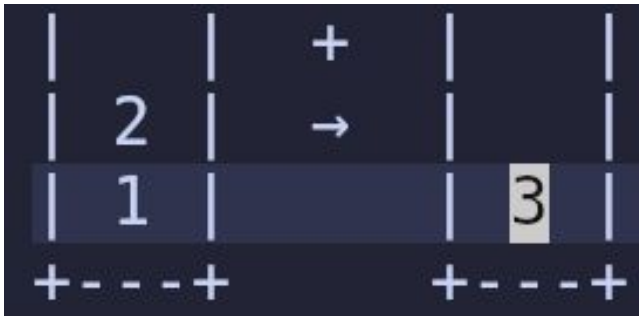
disassemble Func
```



```

Func
  var a = 1
   0 STORE 1 in $0


  echo a + 2
   1 LOAD $0
   2 PUSHNR 2
   3 OPNR +
   4 ECHO 1
   5 RETURN void
```



compiler

- Vim9 script programs are compiled in src/vim9compile.c
  - function is compiled in compile_def_function()
- Optimization
  - constant folding
    - 1+1 -> 2

Enjoy Hacking Vim script !